DEITEL® DEVELOPER SERIES

SECOND EDITION

# Android™ for Programmers

## An App-Driven Approach

Volume 1

7 Fully Coded Android™ Apps

PAUL DEITEL • HARVEY DEITEL
ABBEY DEITEL

# ANDROID™ FOR PROGRAMMERS
## AN APP-DRIVEN APPROACH
## SECOND EDITION, VOLUME 1
## DEITEL® DEVELOPER SERIES

*Library of Congress Cataloging-in-Publication Data*

On file



ISBN-13: 978-0-13357092-2
ISBN-10: 0-13-357092-4

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, December 2013

# Android™ for Programmers
## An App-Driven Approach
## Second Edition, Volume 1
## Deitel® Developer Series

**Paul Deitel • Harvey Deitel • Abbey Deitel**
*Deitel & Associates, Inc.*

PRENTICE HALL

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

## Trademarks

DEITEL, the double-thumbs-up bug and DIVE-INTO are registered trademarks of Deitel & Associates, Inc.

Java is a registered trademark of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Google, Android, Google Play, Google Maps, Google Wallet, Nexus, YouTube, AdSense and AdMob are trademarks of Google, Inc.

Microsoft and/or its respective suppliers make no representations about the suitability of the information contained in the documents and related graphics published as part of the services for any purpose. All such documents and related graphics are provided "as is" without warranty of any kind. Microsoft and/or its respective suppliers hereby disclaim all warranties and conditions with regard to this information, including all warranties and conditions of merchantability, whether express, implied or statutory, fitness for a particular purpose, title and non-infringement. In no event shall Microsoft and/or its respective suppliers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of information available from the services.

The documents and related graphics contained herein could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Microsoft and/or its respective suppliers may make improvements and/or changes in the product(s) and/or the program(s) described herein at any time. Partial screen shots may be viewed in full within the software version specified.

Microsoft® and Windows® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. Screen shots and icons reprinted with permission from the Microsoft Corporation. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

Throughout this book, trademarks are used. Rather than put a trademark symbol in every occurrence of a trademarked name, we state that we are using the names in an editorial fashion only and to the benefit of the trademark owner, with no intention of infringement of the trademark.

*In Memory of Amar G. Bose, MIT Professor and Founder and Chairman of the Bose Corporation:*

*It was a privilege being your student—and members of the next generation of Deitels, who heard our dad say how your classes inspired him to do his best work.*

*You taught us that if we go after the really hard problems, then great things can happen.*

*Harvey Deitel*
*Paul and Abbey Deitel*

*This page intentionally left blank*

# Contents

# Preface

Welcome to the dynamic world of Android smartphone and tablet app development with the Android Software Development Kit (SDK), the Java™ programming language, the Eclipse-based Android Development Tools IDE, and the new and rapidly evolving Android Studio IDE.

*Android for Programmers: An App-Driven Approach, 2/e, Volume 1* presents leading-edge mobile computing technologies for professional software developers. At the heart of the book is our *app-driven approach*—we present concepts in the context of *seven complete working Android apps* rather than using code snippets. Chapters 2–8 each present one app. We begin each of these chapters with an introduction to the app, an app test-drive showing one or more sample executions and a technologies overview. Then we proceed with a detailed code walkthrough of the app's source code. All of the source code is available at `www.deitel.com/books/AndroidFP2`. We recommend that you have the source code open in the IDE as you read the book.

Sales of Android devices and app downloads have been growing exponentially. The first-generation Android phones were released in October 2008. A study by Strategy Analytics showed that by October 2013, Android had 81.3% of the global smartphone market share, compared to 13.4% for Apple, 4.1% for Microsoft and 1% for Blackberry.[1] According to an IDC report, by the end of the first quarter of 2013 Android had 56.5% of the global tablet market share, compared to 39.6% for Apple's iPad and 3.7% for Microsoft Windows tablets.[2]

There are now over one billion Android smartphones and tablets in use,[3] and more than 1.5 million Android devices are being activated daily.[4] According to IDC, Samsung is the leading Android manufacturer, accounting for nearly 40% of Android device shipments in the third quarter of 2013.

Billions of apps have been downloaded from Google Play™—Google's marketplace for Android apps. The opportunities for Android app developers are enormous.

Fierce competition among popular mobile platforms and carriers is leading to rapid innovation and falling prices. Competition among the dozens of Android device manufacturers is driving hardware and software innovation within the Android community.

## Copyright Notice and Code License



1. `http://blogs.strategyanalytics.com/WSS/post/2013/10/31/Android-Captures-Record-81-Percent-Share-of-Global-Smartphone-Shipments-in-Q3-2013.aspx`.
2. `http://www.idc.com/getdoc.jsp?containerId=prUS24093213`.
3. `http://www.android.com/kitkat`.
4. `http://www.technobuffalo.com/2013/04/16/google-daily-android-activations-1-5-million`.

*3.0 Unported License* (`http://creativecommons.org/licenses/by/3.0`), *with the exception that they may not be reused in any way in educational tutorials and textbooks, whether in print or digital format. Additionally, the authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in this book. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs. You're welcome to use the apps in the book as shells for your own apps, building on their existing functionality. If you have any questions, contact us at* `deitel@deitel.com`.

## Intended Audience

We assume that you're a Java programmer with object-oriented programming experience. Because of the improved Android development tools, we were able to eliminate almost all XML markup in this edition. There are still two small, easy-to-understand XML files you'll need to manipulate. We use only complete, working apps, so if you don't know Java but have object-oriented programming experience in languages like C#/.NET, Objective-C/Cocoa or C++ (with class libraries), you should be able to master the material quickly, learning a good amount of Java and Java-style object-oriented programming along the way.

This book is *not* a Java tutorial, but it presents a significant amount of Java in the context of Android app development. If you're interested in learning Java, check out our publications:

- *Java for Programmers, 2/e* (`www.deitel.com/books/javafp2`)
- *Java Fundamentals: Parts I and II* LiveLessons videos (`www.deitel.com/books/LiveLessons`).
- *Java How to Program, 10/e* (`www.deitel.com/books/jhtp10`)

If you're not familiar with XML, see these online tutorials:

- `http://www.ibm.com/developerworks/xml/newto`
- `http://www.w3schools.com/xml/xml_whatis.asp`
- `http://www.deitel.com/articles/xml_tutorials/20060401/XMLBasics`
- `http://www.deitel.com/articles/xml_tutorials/20060401/XMLStructuringData`

## Key Features

Here are some of this book's key features:

***App-Driven Approach.*** Chapters 2–8 each present one completely coded app—we discuss what the app does, show screen shots of the app in action, test-drive it and overview the technologies and architecture we'll use to build it. Then we build the app's GUI and resource files, present the complete code and do a detailed code walkthrough. We discuss the programming concepts and demonstrate the functionality of the Android APIs used in the app.

***Android SDK 4.3 and 4.4.*** We cover various new Android Software Development Kit (SDK) 4.3 and 4.4 features.

***Fragments.*** Starting with Chapter 5, we use `Fragments` to create and manage portions of each app's GUI. You can combine several fragments to create user interfaces that take ad-

vantage of tablet screen sizes. You also can easily interchange fragments to make your GUIs more dynamic, as you'll do in Chapter 8.

***Support for multiple screen sizes and resolutions.*** Throughout the app chapters we demonstrate how to use Android's mechanisms for automatically choosing resources (layouts, images, etc.) based on a device's size and orientation.

***Eclipse-Based Android Development Tools (ADT) IDE coverage in the print book.*** The free Android Development Tools (ADT) integrated development environment (IDE)—which includes Eclipse and the ADT plugin—combined with the free Java Development Kit (JDK) provide all the software you'll need to create, run and debug Android apps, export them for distribution (e.g., upload them to Google Play™) and more.

***Android Studio IDE.*** This is the preferred IDE for the future of Android app development. Because it's new and evolving rapidly, we put our discussions of it online at:

```
http://www.deitel.com/books/AndroidFP2
```

We'll show how to import existing projects so you can test-drive our apps. We'll also demonstrate how to create new apps, build GUIs, modify resource files and test your apps. If you have any questions, contact us at `deitel@deitel.com`.

***Immersive Mode.*** The status bar at the top of the screen and the menu buttons at the bottom can be hidden, allowing your apps to fill more of the screen. Users can access the status bar by swiping down from the top of the screen, and the system bar (with the back button, home button and recent apps button) by swiping up from the bottom.

***Printing Framework.*** Android 4.4 KitKat allows you to add printing functionality to your apps, such as locating available printers over Wi-Fi or the cloud, selecting the paper size and specifying which pages to print.

***Testing on Android Smartphones, Tablets and the Android Emulator.*** For the best app-development experience, you should test your apps on actual Android smartphones and tablets. You can still have a meaningful experience using just the Android emulator (see the Before You Begin section), however it's processor-intensive and can be slow, particularly with games that have a lot of moving parts. In Chapter 1, we mention some Android features that are not supported on the emulator.

***Multimedia.*** The apps use a broad range of Android multimedia capabilities, including graphics, images, frame-by-frame animation and audio.

***Uploading Apps to Google Play.*** Chapter 9, Google Play and App Business Issues, walks you through the registration process for Google Play and setting up a merchant account so you can sell your apps. You'll learn how to prepare apps for submission to Google Play, find tips for pricing your apps, and resources for monetizing them with in-app advertising and in-app sales of virtual goods. You'll also find resources for marketing your apps. Chapter 9 can be read after Chapter 1.

## Features

***Syntax Coloring.*** For readability, we syntax color the code, similar to Eclipse's and Android Studio's use of syntax coloring. Our syntax-coloring conventions are as follows:

```
comments appear in green
keywords appear in dark blue
constants and literal values appear in light blue
all other code appears in non-bold black
```

*Code Highlighting.* We emphasize the key code segments in each program by enclosing them in yellow rectangles.

*Using Fonts for Emphasis.* We use various font conventions:

- The defining occurrences of key terms appear in **bold maroon** for easy reference.
- On-screen IDE components appear in **bold Helvetica** (e.g., the **File** menu).
- Program source code appears in `Lucida` (e.g., `int x = 5;`).

In this book you'll create GUIs using a combination of visual programming (point and click, drag and drop) and writing code.

We use different fonts when we refer to GUI elements in program code versus GUI elements displayed in the IDE:

- When we refer to a GUI component that we create in a program, we place its class name and object name in a `Lucida` font—e.g., "`Button saveContactButton`."
- When we refer to a GUI component that's part of the IDE, we place the component's text in a **bold Helvetica** font and use a plain text font for the component's type—e.g., "the **File** menu" or "the **Run** button."

*Using the > Character.* We use the **>** character to indicate selecting a menu item from a menu. For example, we use the notation **File > New** to indicate that you should select the **New** menu item from the **File** menu.

*Source Code.* All of the book's source code is available for download from:

```
www.deitel.com/books/AndroidFP2
www.informit.com/title/0133570924
```

*Documentation.* All the Android and Java documentation you'll need to develop Android apps is available free at `http://developer.android.com` and `http://www.oracle.com/technetwork/java/javase/downloads/index.html`. The documentation for Eclipse is available at `www.eclipse.org/documentation`. The documentation for Android Studio is available at `http://developer.android.com/sdk/installing/studio.html`.

*Chapter Objectives.* Each chapter begins with a list of learning objectives.

*Figures.* Hundreds of tables, source code listings and Android screen shots are included.

*Software Engineering*. We stress program clarity and performance, and concentrate on building well-engineered, object-oriented software.

*Index.* We include an extensive index for reference. The page number of the defining occurrence of each key term in the book is highlighted in the index in **bold maroon**.

## Working with Open-Source Apps

There are numerous free, open-source Android apps available online which are excellent resources for learning Android app development. We encourage you to download open-

source apps and read their source code to understand how they work. **Caution: The terms of open-source licenses vary considerably.** Some allow you to use the app's source code freely for any purpose, while others stipulate that the code is available for personal use only—not for creating for-sale or publicly available apps. **Be sure to read the licensing agreements carefully. If you wish to create a commercial app based on an open-source app, you should consider having an intellectual property attorney read the license; be aware that these attorneys charge significant fees.**

## *Android for Programmers: An App-Driven Approach, Second Edition, Volume 2*

Volume 2, which will be published in 2014, contains additional app-development chapters that introduce property animation, Google Play game services, video, speech synthesis and recognition, GPS, the Maps API, the compass, object serialization, web services, audio recording and playback, Bluetooth®, HTML5 mobile apps and more. **For the status of Volume 2 and for continuing book updates, visit**

```
http://www.deitel.com/books/AndroidFP2
```

## *Android Fundamentals, Second Edition* LiveLessons Video Training Products

Our *Android Fundamentals, Second Edition* LiveLessons videos show you what you need to know to start building robust, powerful Android apps with the Android Software Development Kit (SDK) 4.3 and 4.4, the Java™ programming language and the Eclipse™ and Android Studio integrated development environments (IDEs). It will include approximately 20 hours of expert training synchronized with *Android for Programmers, Second Edition* (Volumes 1 and 2). The videos for Volume 1 will be available spring 2014. For additional information about Deitel LiveLessons video products, visit

```
www.deitel.com/livelessons
```

or contact us at `deitel@deitel.com`. You can also access our LiveLessons videos if you have a subscription to Safari Books Online (`www.safaribooksonline.com`).

## Join the Deitel & Associates, Inc. Social Networking Communities

To receive updates on this and our other publications, new and updated apps, online Resource Centers, instructor-led onsite training courses, partner offers and more, join the Deitel social networking communities on Facebook® (`http://www.deitel.com/deitelfan`), Twitter® (`@deitel`), LinkedIn® (`http://bit.ly/DeitelLinkedIn`) Google+™ (`http://google.com/+DeitelFan`), and YouTube® (`http://youtube.com/user/DeitelTV`) and subscribe to the *Deitel® Buzz Online* newsletter (`http://www.deitel.com/newsletter/subscribe.html`).

## Contacting the Authors

We'd sincerely appreciate your comments, criticisms, corrections and suggestions for improvement. Please address all questions and other correspondence to:

```
deitel@deitel.com
```

We'll respond promptly, and post corrections and clarifications on:

```
www.deitel.com/books/AndroidFP2
```

and on Facebook, Twitter, Google+, LinkedIn and the *Deitel® Buzz Online*.

## Visit www.deitel.com to:

- Download code examples
- Check out the growing list of programming Resource Centers
- Receive updates for this e-book, subscribe to the free *Deitel® Buzz Online* e-mail newsletter at `www.deitel.com/newsletter/subscribe.html`
- Receive information on our *Dive Into® Series* instructor-led programming language training courses offered at customer sites worldwide

## Acknowledgments

Thanks to Barbara Deitel for long hours devoted to this project—she created all of our Android Resource Centers, and patiently researched hundreds of technical details.

This book was a cooperative effort between professional and academic divisions of Pearson. We appreciate the efforts and 18-year mentorship of our friend and professional colleague Mark L. Taub, Editor-in-Chief of the Pearson Technology Group. Mark and his team handle all of our professional books and LiveLessons video products. Kim Boedigheimer recruited distinguished members of the Android community and managed the review team for the Android content. We selected the cover art and Chuti Prasertsith and Sandra Schroeder designed the cover. John Fuller manages the production of all of our Deitel Developer Series books.

We also appreciate the guidance, wisdom and energy of Tracy Johnson, Executive Editor, Computer Science. Tracy and her team handle all of our academic textbooks. Carole Snyder recruited the book's academic reviewers and managed the review process. Bob Engelhardt manages the production of our academic publications.

We'd like to thank Michael Morgano, a former colleague of ours at Deitel & Associates, Inc., now an Android developer at Imerj™, who co-authored the first editions of this book and our book, *iPhone for Programmers: An App-Driven Approach*. Michael is an extraordinarily talented software developer.

### *Reviewers of the Content from* Android for Programmers: An App-Driven Approach *and* Android How to Program *Recent Editions*

We wish to acknowledge the efforts of our first and second edition reviewers. They scrutinized the text and the code and provided countless suggestions for improving the presentation: Paul Beusterien (Principal, Mobile Developer Solutions), Eric J. Bowden, COO (Safe Driving Systems, LLC), Tony Cantrell (Georgia Northwestern Technical College), Ian G. Clifton (Independent Contractor and Android App Developer, Daniel Galpin (Android Advocate and author of *Intro to Android Application Development*), Jim Hathaway (Application Developer, Kellogg Company), Douglas Jones (Senior Software Engineer, Fullpower Technologies), Charles Lasky (Nagautuck Community College), Enrique Lopez-Manas (Lead Android Architect, Sixt, and Computer Science Teacher at the Univer-

sity of Alcalá in Madrid), Sebastian Nykopp (Chief Architect, Reaktor), Michael Pardo (Android Developer, Mobiata), Ronan "Zero" Schwarz (CIO, OpenIntents), Arijit Sengupta (Wright State University), Donald Smith (Columbia College), Jesus Ubaldo Quevedo-Torrero (University of Wisconsin, Parkside), Dawn Wick (Southwestern Community College) and Frank Xu (Gannon University).

Well, there you have it! *Android for Programmers: An App-Driven Approach, Second Edition, Volume 1* will quickly get you developing Android apps. We hope you enjoy reading the book as much as we enjoyed writing it!

*Paul Deitel*
*Harvey Deitel*
*Abbey Deitel*

## About the Authors

**Paul Deitel**, CEO and Chief Technical Officer of Deitel & Associates, Inc., is a graduate of MIT, where he studied Information Technology. He holds the Java Certified Programmer and Java Certified Developer certifications, and is an Oracle Java Champion. Through Deitel & Associates, Inc., he has delivered hundreds of programming courses worldwide to clients, including Cisco, IBM, Siemens, Sun Microsystems, Dell, Fidelity, NASA at the Kennedy Space Center, the National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, SunGard Higher Education, Nortel Networks, Puma, iRobot, Invensys and many more. He and his co-author, Dr. Harvey M. Deitel, are the world's best-selling programming-language textbook/professional book/video authors.

**Dr. Harvey Deitel**, Chairman and Chief Strategy Officer of Deitel & Associates, Inc., has more than 50 years of experience in computing. Dr. Deitel earned B.S. and M.S. degrees in Electrical Engineering from MIT and a Ph.D. in Mathematics from Boston University. In the 1960s, through Advanced Computer Techniques and Computer Usage Corporation, he worked on the teams building various IBM operating systems. In the 1970s, he built commercial software systems. He has extensive college teaching experience, including earning tenure and serving as the Chairman of the Computer Science Department at Boston College before founding Deitel & Associates, Inc., in 1991 with his son, Paul Deitel. The Deitels' publications have earned international recognition, with translations published in Simplified Chinese, Traditional Chinese, Korean, Japanese, German, Russian, Spanish, French, Polish, Italian, Portuguese, Greek, Urdu and Turkish. Dr. Deitel has delivered hundreds of programming courses to corporate, academic, government and military clients.

**Abbey Deitel**, President of Deitel & Associates, Inc., is a graduate of Carnegie Mellon University's Tepper School of Management where she received a B.S. in Industrial Management. Abbey has been managing the business operations of Deitel & Associates, Inc. for 16 years. She has contributed to numerous Deitel & Associates publications and, together with Paul and Harvey, is the co-author of *Android for Programmers: An App-Driven Approach, 2/e*, *iPhone for Programmers: An App-Driven Approach*, *Internet & World Wide Web How to Program, 5/e*, *Visual Basic 2012 How to Program, 6/e* and *Simply Visual Basic 2010, 5/e*.

## Deitel® Dive-Into® Series Corporate Training

Deitel & Associates, Inc., founded by Paul Deitel and Harvey Deitel, is an internationally recognized authoring and corporate training organization, specializing in Android and iOS app development, computer programming languages, object technology and Internet and web software technology. The company's clients include many of the world's largest corporations, government agencies, branches of the military, and academic institutions. The company offers instructor-led training courses delivered at client sites worldwide on major programming languages and platforms, including Android app development, Objective-C and iOS app development, Java™, C++, Visual C++®, C, Visual C#®, Visual Basic®, XML®, Python®, object technology, Internet and web programming and a growing list of additional programming and software development courses.

Through its 37-year publishing partnership with Prentice Hall/Pearson, Deitel & Associates, Inc., publishes leading-edge programming professional books, college textbooks and *LiveLessons* video courses. Deitel & Associates, Inc. and the authors can be reached at:

```
deitel@deitel.com
```

To learn more about Deitel's *Dive-Into® Series* Corporate Training curriculum, visit:

```
www.deitel.com/training
```

To request a proposal for worldwide on-site, instructor-led training at your organization, e-mail `deitel@deitel.com`.

Individuals wishing to purchase Deitel books and *LiveLessons* video training can do so through `www.deitel.com`. Bulk orders by corporations, the government, the military and academic institutions should be placed directly with Pearson. For more information, visit

```
www.informit.com/store/sales.aspx
```

# Before You Begin

In this section, you'll set up your computer for use with this book. The Android development tools are frequently updated. Before reading this section, check the book's website

```
http://www.deitel.com/books/AndroidFP2/
```

to see if we've posted an updated version.

## Font and Naming Conventions

We use fonts to distinguish between on-screen components (such as menu names and menu items) and Java code or commands. Our convention is to show on-screen components in a sans-serif bold **Helvetica** font (for example, **Project** menu) and to show file names, Java code and commands in a sans-serif `Lucida` font (for example, the keyword `public` or class `Activity`). When specifying commands to select in menus, we use the **>** notation to indicate a menu item to select. For example, **Window > Preferences** indicates that you should select the **Preferences** menu item from the **Window** menu.

## Software and Hardware System Requirements

To develop Android apps you need a Windows®, Linux or Mac OS X system. To view the latest operating-system requirements visit:

```
http://developer.android.com/sdk/index.html
```

and scroll down to the **SYSTEM REQUIREMENTS** heading. We developed the apps in this book using the following software:

- Java SE 7 Software Development Kit
- Android SDK/ADT Bundle based on the Eclipse IDE
- Android SDK versions 4.3 and 4.4

You'll see how to obtain each of these in the next sections.

## Installing the Java Development Kit (JDK)

Android requires the *Java Development Kit* (*JDK*) version 7 (JDK 7) or 6 (JDK 6). *We used JDK 7*. To download the JDK for Windows, OS X or Linux, go to

```
http://www.oracle.com/technetwork/java/javase/downloads/index.html
```

You need only the JDK. Choose the 32-bit or 64-bit version based on your computer hardware and operating system. Most recent computers have 64-bit hardware—check your system's specifications. If you have a 32-bit operating system, you must use the 32-bit JDK. Be sure to follow the installation instructions at

```
http://docs.oracle.com/javase/7/docs/webnotes/install/index.html
```

## Android Integrated Development Environment (IDE) Options

Google now provides two Android IDE options:

- Android SDK/ADT bundle—a version of the *Eclipse IDE* that comes preconfigured with the latest Android Software Development Kit (SDK) and the latest Android Development Tools (ADT) plugin. At the time of this writing, these were Android SDK version 4.4 and ADT version 22.3.
- Android Studio—Google's new Android IDE based on IntelliJ® IDEA and their preferred future IDE.

The Android SDK/ADT bundle has been widely used in Android app development for several years. Android Studio, introduced in May 2013, is an *early access version* and will be evolving rapidly. For this reason, we'll stay with the widely used Android SDK/ADT bundle in the book, and as online supplements at

```
http://www.deitel.com/books/AndroidFP2
```

we'll provide Android Studio versions of the Chapter 1 Test-Drive section and the Building the GUI section for each app, as appropriate.

## Installing the Android SDK/ADT Bundle

To download the Android SDK/ADT bundle, go to

```
http://developer.android.com/sdk/index.html
```

and click the **Download the SDK ADT Bundle** button. When the download completes, extract the ZIP file's contents to your system. The resulting folder has an `eclipse` subfolder containing the Eclipse IDE and an `sdk` subfolder containing the Android SDK. As with the JDK, you can choose a 32-bit or 64-bit version. The Android SDK/ADT bundle 32-bit version should be used with the 32-bit JDK, and the 64-bit version with the 64-bit JDK.

## Installing Android Studio

The IDE instructions in the printed book use the Android SDK/ADT bundle. You can also optionally install and use Android Studio. To download Android Studio, go to

```
http://developer.android.com/sdk/installing/studio.html
```

and click the **Download Android Studio** button. When the download completes, run the installer and follow the on-screen instructions to complete the installation. [*Note:* For Android 4.4 development in Android Studio, Android now supports Java SE 7 language features, including the diamond operator, multi-`catch`, `Strings` in `switch` and `try`-with-resources.]

## Set the Java Compiler Compliance Level and Show Line Numbers

*Android does not fully support Java SE 7.* To ensure that the book's examples compile correctly, configure Eclipse to produce files that are compatible with Java SE 6 by performing the following steps:

1. Open Eclipse ( or ), which is located in the `eclipse` subfolder of the Android SDK/ADT bundle's installation folder.
2. When the **Workspace Launcher** window appears, click **OK**.

3. Select **Window > Preferences** to display the **Preferences** window. On Mac OS X, select **ADT > Preferences...**.
4. Expand the **Java** node and select the **Compiler** node. Under **JDK Compliance**, set the **Compiler compliance level** to 1.6 (to indicate that Eclipse should produce compiled code that's compatible with Java SE 6).
5. Expand the **General > Editors** node and select **TextEditors**, then ensure that **Show line numbers** is selected and click **OK**.
6. Close Eclipse.

## Android 4.3 SDK

This book's examples were written using the Android 4.3 and 4.4 SDKs. At the time of this writing, 4.4 was the version included with the Android SDK/ADT bundle and Android Studio. You should also install Android 4.3 (and any other versions you might want to support in your apps). To install other Android platform versions, perform the following steps (skipping Steps 1 and 2 if Eclipse is already open):

1. Open Eclipse. Depending on your platform, the icon will appear as  or  .
2. When the **Workspace Launcher** window appears, click **OK**.
3. On Mac OS X, if you see a window indicating "**Could not find SDK folder '/Users/***YourAccount*/**android-sdk-macosx/'**," click **Open Preferences** then **Browse...** and select the `sdk` folder located where you extracted the Android SDK/ADT bundle.
4. Select **Window > Android SDK Manager** to display the **Android SDK Manager** (Fig. 1).

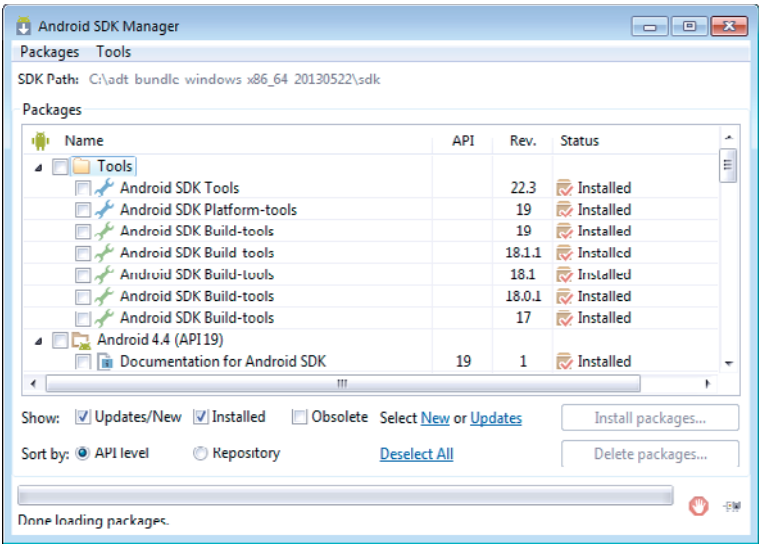



**Fig. 1** | **Android SDK Manager** window.

5. The **Android SDK Manager**'s **Name** column shows all of the tools, platform versions and extras (such as APIs for interacting with Google services, like Maps) that you

can install. Uncheck the **Installed** checkbox. Then, if any of **Tools**, **Android 4.4 (API19)**, **Android 4.3 (API18)** and **Extras** appear in the **Packages** list, ensure that they're checked and click **Install # packages...** (# is the number of items to be installed) to display the **Choose Packages to Install** window. Most items in the **Extras** node are optional. For this book, you'll need the **Android Support Library** and **Google Play services**. The **Google USB Driver** is necessary for Windows users who wish to test apps on Android devices.]

6. In the **Choose Packages to Install** window, read the license agreements for each item. When you're done, click the **Accept License** radio button, then click the **Install** button. The status of the installation process will be displayed in the **Android SDK Manager** window.

## Creating Android Virtual Devices (AVDs)

The **Android emulator**, included in the Android SDK, allows you to test apps on your computer rather than on an actual Android device. This is useful if you're learning Android and don't have access to Android devices, but can be *very* slow, so a real device is preferred if you have one. There are some hardware acceleration features that can improve emulator performance (`developer.android.com/tools/devices/emulator.html#acceleration`). Before running an app in the emulator, you must create an **Android Virtual Device (AVD)** which defines the characteristics of the device you want to test on, including the screen size in pixels, the pixel density, the physical size of the screen, size of the SD card for data storage and more. To test your apps for multiple Android devices, you can create AVDs that emulate each unique device. For this book, we use AVDs for Google's Android reference devices—the Nexus 4 phone, the Nexus 7 small tablet and Nexus 10 large tablet—which run unmodified versions of Android. To do so, perform the following steps:

1. Open Eclipse.
2. Select **Window > Android Virtual Device Manager** to display the **Android Virtual Device Manager** window, then select the **Device Definitions** tab (Fig. 2).

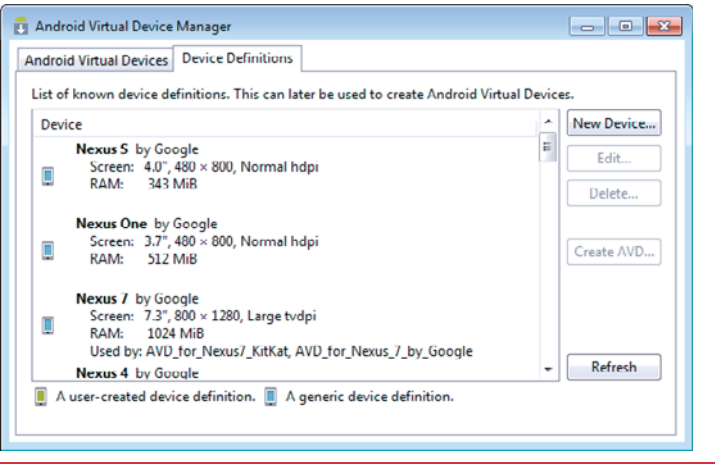



**Fig. 2** | **Android Virtual Device Manager** window.

3. Google provides preconfigured devices that you can use to create AVDs. Select **Nexus 4 by Google**, then click **Create AVD...** to display the **Create new Android Virtual Device (AVD)** window (Fig. 3), then configure the options as shown and click **OK** to create the AVD. If you check **Hardware keyboard present**, you'll be able to use your computer's keyboard to type data into apps that are running in the AVD, but this may prevent the soft keyboard from displaying on the screen. If your computer does not have a camera, you can select **Emulated** for the **Front Camera** and **Back Camera** options. Each AVD you create has many other options specified in its `config.ini`. You can modify this file as described at

```
http://developer.android.com/tools/devices/managing-avds.html
```

to more precisely match the hardware configuration of your device.

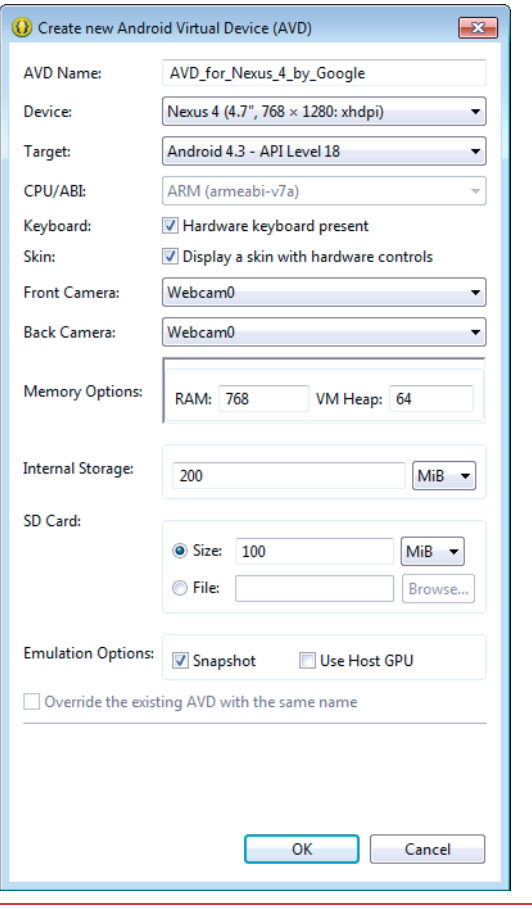



**Fig. 3** | Configuring a Nexus 4 smartphone AVD for Android 4.3.

4. We also configured Android 4.3 AVDs that represent Nexus 7 by Google and Nexus 10 by Google for testing our tablet apps. Their settings are shown in Fig. 4. In

addition, we configured Android 4.4 AVDs for the Nexus 4, Nexus 7 and Nexus 10 with the names: `AVD_for_Nexus_4_KitKat`, `AVD_for_Nexus_7_KitKat`, and `AVD_for_Nexus_10_KitKat`,

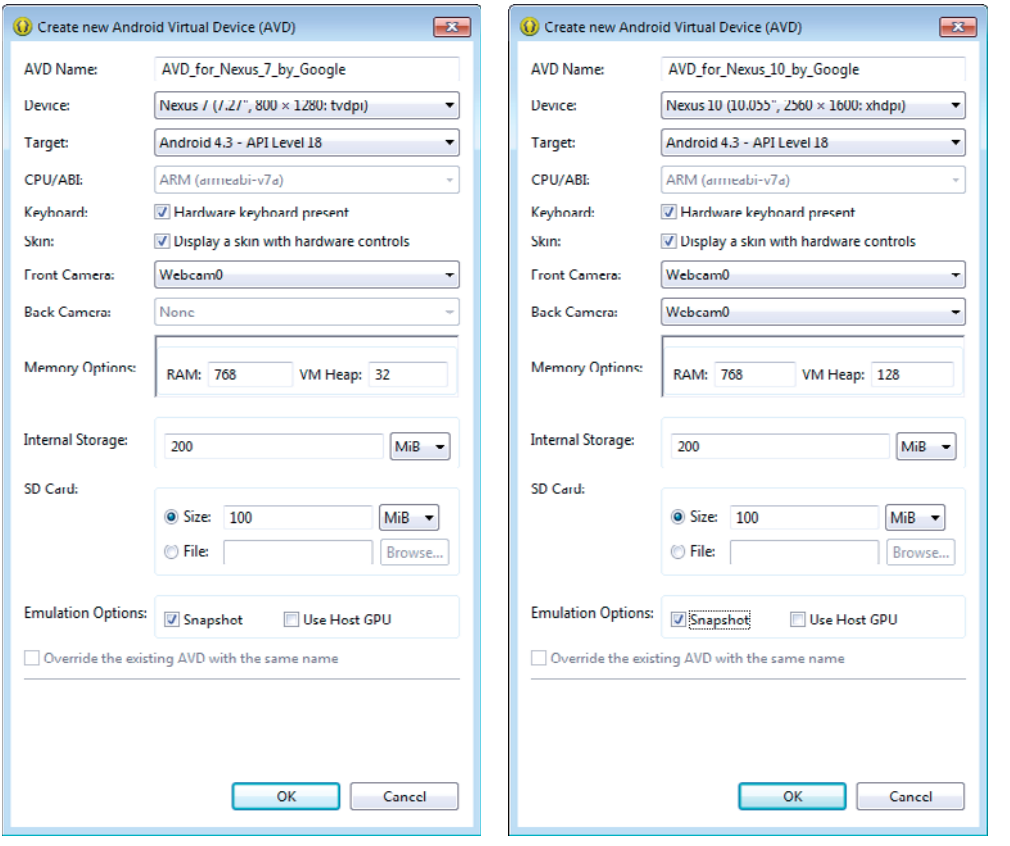



**Fig. 4** | Configuring Nexus 7 and Nexus 10 tablet AVDs.

## (Optional) Setting Up an Android Device for Development

As we mentioned, testing apps on AVDs can be slow due to AVD performance. If you have an Android device available to you, you should test the apps on that device. In addition, there are some features that you can test only on actual devices. To execute your apps on Android devices, follow the instructions at

```
http://developer.android.com/tools/device.html
```

If you're developing on Microsoft Windows, you'll also need the Windows USB driver for Android devices. In some cases on Windows, you may also need device-specific USB drivers. For a list of USB driver sites for various device brands, visit:

```
http://developer.android.com/tools/extras/oem-usb.html
```

## Obtaining the Book's Code Examples

The examples for *Android for Programmers, 2/e, Volume 1* are available for download at

```
http://www.deitel.com/books/AndroidFP2/
```

If you're not already registered at our website, go to `www.deitel.com` and click the **Register** link. Fill in your information. Registration is free, and we do not share your information with anyone. Please verify that you entered your registration e-mail address correctly—you'll receive a confirmation e-mail with your verification code. *You must click the verification link in the e-mail before you can sign in at `www.deitel.com` for the first time*. Configure your e-mail client to allow e-mails from `deitel.com` to ensure that the verification e-mail is not filtered as junk mail. We send only occasional account-management e-mails unless you register separately for our free *Deitel® Buzz Online* e-mail newsletter at

```
http://www.deitel.com/newsletter/subscribe.html
```

Next, visit `www.deitel.com` and sign in using the **Login** link below our logo in the upper-left corner of the page. Go to `http://www.deitel.com/books/AndroidFP2/`. Click the **Examples** link to download a ZIP archive file containing the examples to your computer. Double click the ZIP file to unzip the archive, and make note of where you extract the file's contents on your system.

## A Note Regarding the Android Development Tools

Google *frequently* updates the Android development tools. This often leads to problems compiling our apps when, in fact, the apps do not contain any errors. If you import one of our apps into Eclipse or Android Studio and it does not compile, there is probably a minor configuration issue. Please contact us by e-mail at `deitel@deitel.com` or by posting a question to:

- Facebook®—`facebook.com/DeitelFan`
- Google+™—`google.com/+DeitelFan`

and we'll help you resolve the issue.

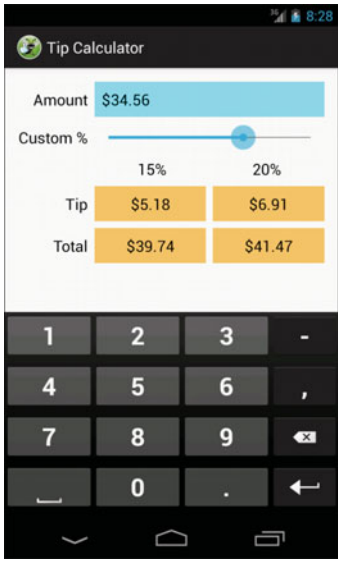
You've now installed all the software and downloaded the code examples you'll need to study Android app development with *Android for Programmers, 2/e, Volume 1* and to begin developing your own apps. Enjoy!

*This page intentionally left blank*

# 3

# Tip Calculator App

## Introducing `GridLayout`, `LinearLayout`, `EditText`, `SeekBar`, Event Handling, `NumberFormat` and Defining App Functionality with Java




### Objectives

In this chapter you'll:

- Design a GUI using `LinearLayout`s and a `GridLayout`.
- Use the IDE's **Outline** window to add GUI components to `LinearLayout`s and a `GridLayout`.
- Use `TextView`, `EditText` and `SeekBar` GUI components.
- Use Java object-oriented programming capabilities, including classes, objects, interfaces, anonymous inner classes and inheritance to add functionality to an Android app.
- Programmatically interact with GUI components to change the text that they display.
- Use event handling to respond to user interactions with an `EditText` and a `SeekBar`.
- Specify that the keypad should always be displayed when an app is executing.
- Specify that an app supports only portrait orientation.

Outline



## 3.1 Introduction

The **Tip Calculator** app (Fig. 3.1(a)) calculates and displays possible tips for a restaurant bill. As you enter each digit of a bill amount by touching the *numeric keypad*, the app calculates and displays the tip amount and total bill (bill amount + tip) for a 15% tip and a custom

a) Initial GUI

b) GUI after user enters the amount 34.56 and changes the custom tip percentage to 20%

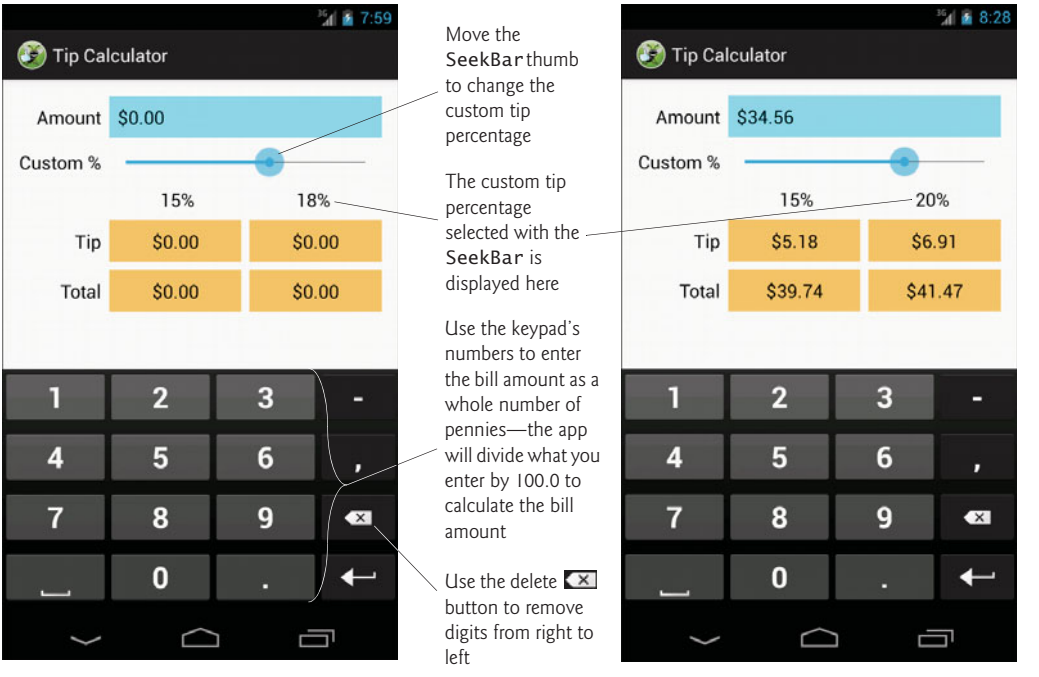



**Fig. 3.1** | Entering the bill total and calculating the tip.

tip percentage (18% by default). You can specify a custom tip percentage from 0% to 30% by moving the `SeekBar` *thumb*—this updates the custom percentage shown and displays the custom tip and total (Fig. 3.1(b)). We chose 18% as the default custom percentage, because many restaurants in the United States add this tip percentage for parties of six people or more. The keypad in Fig. 3.1 may differ based on your AVD's or device's Android version, or based on whether you've installed and selected a custom keyboard on your device.

You'll begin by test-driving the app—you'll use it to calculate 15% and custom tips. Then we'll overview the technologies you'll use to create the app. You'll build the app's GUI using the Android Developer Tools IDE's **Graphical Layout** editor and the **Outline** window. Finally, we'll present the complete Java code for the app and do a detailed code walkthrough. We provide online an Android Studio version of Sections 3.2 and 3.4 at `http://www.deitel.com/books/AndroidFP2`.

## 3.2 Test-Driving the Tip Calculator App

### *Opening and Running the App*

Open the Android Developer Tools IDE and import the **Tip Calculator** app project. Perform the following steps:

1. ***Launching the Nexus 4 AVD.*** For this test-drive, we'll use the Nexus 4 smartphone AVD that you configured in the Before You Begin section. To launch the Nexus 4 AVD, select **Window > Android Virtual Device Manager** to display the **Android Virtual Device Manager** dialog. Select the Nexus 4 AVD and click **Start...**, then click the **Launch** button in the **Launch Options** dialog that appears.
2. ***Opening the Import Dialog.*** Select **File > Import...** to open the **Import** dialog.
3. ***Importing the Tip Calculator app's project.*** Expand the **General** node, select **Existing Projects into Workspace**, then click **Next >** to proceed to the **Import Projects** step. Ensure that **Select root directory** is selected, then click **Browse...**. In the **Browse For Folder** dialog, locate the `TipCalculator` folder in the book's examples folder, select it and click **OK**. Ensure that **Copy projects into workspace** is *not* selected. Click **Finish** to import the project. It now appears in the **Package Explorer** window.
4. ***Launching the Tip Calculator app.*** Right click the `TipCalculator` project in the **Package Explorer** window, then select **Run As > Android Application** to execute **Tip Calculator** in the AVD.

### *Entering a Bill Total*

Using the numeric keypad, enter `34.56`. Just type `3456`—the app will position the cents to the right of the decimal point. If you make a mistake, press the delete (⌫) button to erase one rightmost digit at a time. The `TextViews` under the **15%** and the custom tip percentage (**18%** by default) labels show the tip amount and the total bill for these tip percentages. All the **Tip** and **Total** `TextViews` update each time you enter or delete a digit.

### *Selecting a Custom Tip Percentage*

Use the `Seekbar` to specify a *custom* tip percentage. Drag the `Seekbar`'s *thumb* until the custom percentage reads **20%** (Fig. 3.1(b)). As you drag the thumb, the tip and total for this custom tip percentage update continuously. By default, the `Seekbar` allows you to select values from 0 to 100, but we specified a maximum value of 30 for this app.

## 3.3 Technologies Overview

This section introduces the IDE features and Android technologies you'll use to build the **Tip Calculator** app. We assume that you're *already* familiar with Java object-oriented programming. You'll:

- use various Android classes to create objects
- call methods on Android classes and objects
- define and call your own methods
- use inheritance to create a subclass of Android's `Activity` class that defines the **Tip Calculator**'s functionality
- use event handling, anonymous inner classes and interfaces to process the user's GUI interactions

### 3.3.1 Class `Activity`

Unlike many Java apps, Android apps *don't have a `main` method*. Instead, they have four types of executable components—*activities*, *services*, *content providers* and *broadcast receivers*. In this chapter, we'll discuss activities, which are defined as subclasses of **`Activity`** (package **`android.app`**). Users interact with an `Activity` through *views*—that is, GUI components. Before Android 3.0, a separate `Activity` was typically associated with each screen of an app. As you'll see, starting in Chapter 5, an `Activity` can manage multiple `Fragments`. On a phone, each `Fragment` typically occupies the entire screen and the `Activity` switches between the `Fragments` based on user interactions. On a tablet, activities often display multiple `Fragments` per screen to take better advantage of the larger screen size.

### 3.3.2 `Activity` Lifecycle Methods

Throughout its life, an `Activity` can be in one of several *states*—*active* (i.e., *running*), *paused* or *stopped*. The `Activity` transitions between these states in response to various *events*:

- An *active* `Activity` is *visible* on the screen and "has the focus"—that is, it's in the *foreground*. This is the `Activity` the user is interacting with.
- A *paused* `Activity` is *visible* on the screen but *does not* have the focus—such as when an alert dialog is displayed.
- A *stopped* activity is *not visible* on the screen and is likely to be killed by the system when its memory is needed. An `Activity` is *stopped* when another `Activity` becomes *active*.

As an `Activity` transitions among these states, the Android runtime calls various `Activity` *lifecycle methods*—all of which are defined in the `Activity` class

```
http://developer.android.com/reference/android/app/Activity.html
```

You'll override the **`onCreate`** method in *every* activity. This method is called by the Android runtime when an `Activity` is *starting*—that is, when its GUI is about to be displayed so that the user can interact with the `Activity`. Other lifecycle methods include `onStart`, `onPause`, `onRestart`, `onResume`, `onStop` and `onDestroy`. We'll discuss *most* of these in later chapters. Each activity lifecycle method you override *must* call the superclass's

version; otherwise, an *exception* will occur. This is required because each lifecycle method in superclass `Activity` contains code that must execute in addition to the code you define in your overridden lifecycle methods.

### 3.3.3 Arranging Views with `LinearLayout` and `GridLayout`

Recall that layouts arrange views in a GUI. A **`LinearLayout`** (package **`android.widget`**) arranges views either *horizontally* (the default) or *vertically* and can size its views proportionally. We'll use this to arrange two `TextViews` horizontally and ensure that each uses half of the available horizontal space.

**`GridLayout`** (package `android.widget`) was introduced in Android 4.0 as a new layout for arranging views into cells in a rectangular grid. Cells can occupy *multiple* rows and columns, allowing for complex layouts. In many cases, `GridLayout` can be used to replace the older, and sometimes less efficient `TableLayout`, which arranges views into rows and columns where each row is typically defined as a `TableRow` and the number of columns is defined by the `TableRow` containing the most cells. Normally, `GridLayout` requires API level 14 or higher. However, the *Android Support Library* provides alternate versions of `GridLayout` and many other GUI features so that you can use them in older Android versions. For more information on this library and how to use it in your apps, visit:

```
http://developer.android.com/tools/support-library/index.html
```

A `GridLayout` *cannot* specify within a given row that the horizontal space should be allocated *proportionally* between multiple views. For this reason, several rows in this app's GUI will place two `TextViews` in a horizontal `LinearLayout`. This will enable you to place two `TextViews` in the same `GridLayout` cell and divide the cell's space evenly between them. We'll cover more layouts and views in later chapters—for a complete list, visit:

```
http://developer.android.com/reference/android/widget/
   package-summary.html
```

### 3.3.4 Creating and Customizing the GUI with the Graphical Layout Editor and the Outline and Properties Windows

You'll create `TextViews`, an `EditText` and a `SeekBar` using the IDE's **Graphical Layout** editor (that you used in Chapter 2) and **Outline** window, then customize them with the IDE's **Properties** window—which is displayed at the bottom of the **Outline** window when you're editing a GUI in the **Graphical Layout** editor. You'll do this *without* directly manipulating the XML stored in the files of the project's `res` folder.

An **`EditText`**—often called a *text box* or *text field* in other GUI technologies—is a *subclass* of `TextView` (presented in Chapter 2) that can display text *and* accept text input from the user. You'll specify an `EditText` for *numeric* input, allow users to enter only digits and restrict the *maximum* number of digits that can be entered.

A **`SeekBar`**—often called a *slider* in other GUI technologies—represents an integer in the range 0–100 by default and allows the user to select a number in that range by moving the `SeekBar`'s thumb. You'll customize the `SeekBar` so the user can choose a custom tip percentage *only* from the more limited range 0 to 30.

In the **Properties** window, a view's most commonly customized properties typically appear at the top with their names displayed in bold (Fig. 3.2). All of a view's properties

are also organized into categories within the **Properties** window. For example, class `TextView` inherits many properties from class `View`, so the **Properties** window displays a **TextView** category with `TextView`-specific properties, followed by a **View** category with properties that are inherited from class `View`.

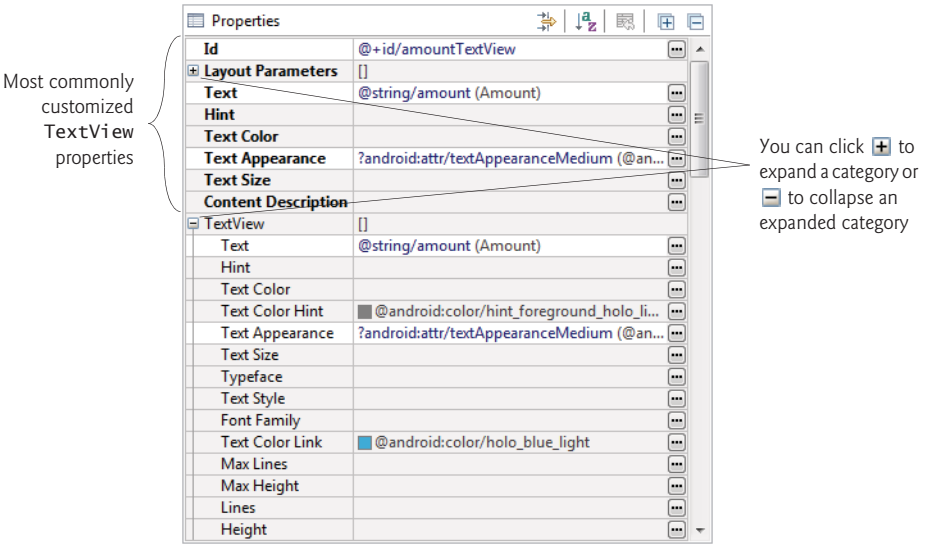



**Fig. 3.2** | **Properties** window showing a `TextView`'s most commonly customized properties.

### 3.3.5 Formatting Numbers as Locale-Specific Currency and Percentage Strings

You'll use class **`NumberFormat`** (package **`java.text`**) to create *locale-specific* currency and percentage strings—an important part of *internationalization*. You could also add *accessibility* strings and internationalize the app using the techniques you learned in Sections 2.7–2.8, though we did not do so in this app.

### 3.3.6 Implementing Interface `TextWatcher` for Handling `EditText` Text Changes

You'll use an *anonymous inner class* to implement the **`TextWatcher`** *interface* (from package **`android.text`**) to respond to *events when the user changes the text* in this app's `EditText`. In particular, you'll use method `onTextChanged` to display the currency-formatted bill amount and to calculate the tip and total as the user enters each digit.

### 3.3.7 Implementing Interface `OnSeekBarChangeListener` for Handling `SeekBar` Thumb Position Changes

You'll implement the `SeekBar.OnSeekBarChangeListener` interface (from package `android.widget`) to respond to the user moving the `SeekBar`'s *thumb*. In particular, you'll

use method `onProgressChanged` to display the custom tip percentage and to calculate the custom tip and total as the user moves the `SeekBar`'s thumb.

### 3.3.8 `AndroidManifest.xml`

The `AndroidManifest.xml` file is created by the IDE when you create a new app project. This file contains many of the settings that you specify in the **New Android Application** dialog, such as the app's name, package name, target and minimum SDKs, `Activity` name(s), theme and more. You'll use the IDE's **Android Manifest** editor to add a new setting to the manifest that forces the *soft keyboard* to remain on the screen. You'll also specify that the app supports only *portrait orientation*—that is, the device's longer side is vertical.

## 3.4 Building the App's GUI

In this section, we'll show the precise steps for building the **Tip Calculator**'s GUI. The GUI will not look like the one shown in Fig. 3.1 until you've completed the steps. As you procede through this section, the number of details presented may seem large, but they're repetitive and you'll get used to them as you use the IDE.

### 3.4.1 `GridLayout` Introduction

This app uses a `GridLayout` (Fig. 3.3) to arrange views into five *rows* and two *columns*. Each cell in a `GridLayout` can be *empty* or can hold one or more *views*, including layouts that *contain* other views. Views can span *multiple* rows or columns, though we did not use that capability in this GUI. You can specify a `GridLayout`'s number of rows and columns in the **Properties** window.

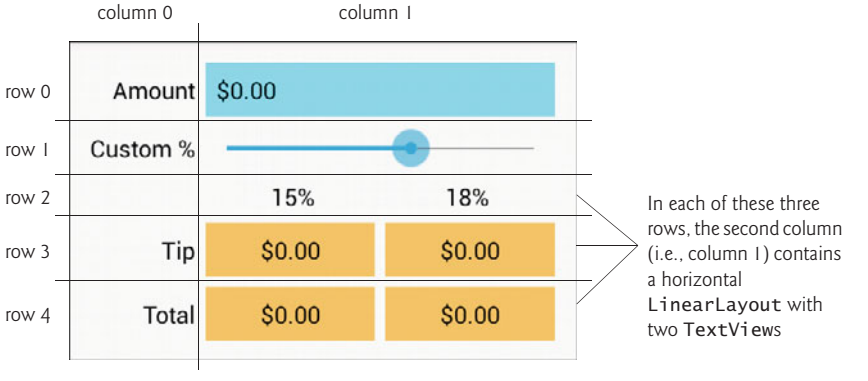



**Fig. 3.3** | **Tip Calculator** GUI's `GridLayout` labeled by its rows and columns.

Each row's *height* is determined by the *tallest* view in that row. Similarly, the *width* of a column is defined by the *widest* view in that column. By default, views are added to a row from left to right. As you'll see, you can specify the exact row and column in which a view is to be placed. We'll discuss other `GridLayout` features as we present the GUI-building steps. To learn more about class `GridLayout`, visit:

```
http://developer.android.com/reference/android/widget/GridLayout.html
```

### *Id Property Values for This App's Views*

Figure 3.4 shows the views' **Id** property values. For clarity, our naming convention is to use the view's class name in the view's **Id** property and Java variable name.

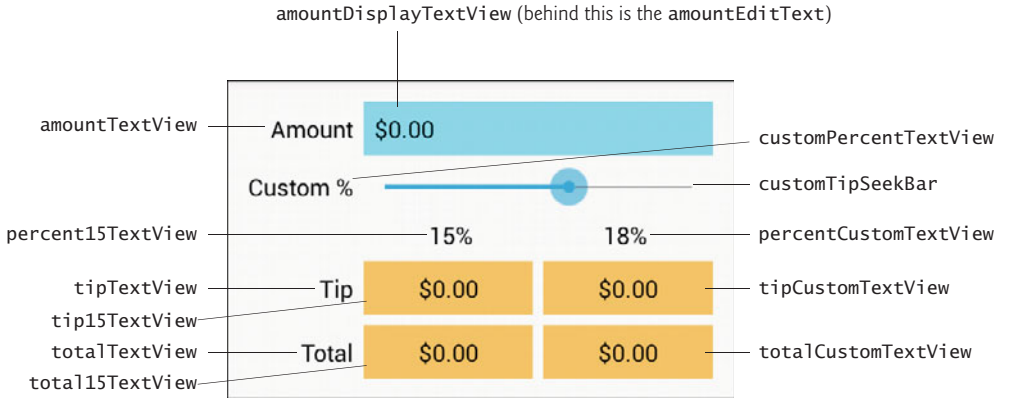



**Fig. 3.4** | **Tip Calculator** GUI's components labeled with their **Id** property values.

In the right column of the first row, there are actually *two* components in the *same* grid cell—the `amountDisplayTextView` is *hiding* the `amountEditText` that receives the user input. As you'll soon see, we restrict the user's input to integer digits so that the user cannot enter invalid input. However, we want the user to see the bill amount as a *currency* value. As the user enters each digit, we divide the amount by 100.0 and display the currency-formatted result in the `amountDisplayTextView`. In the *U.S. locale*, if the user enters 3456, as each digit is entered the `amountDisplayTextView` will show the values `$0.03`, `$0.34`, `$3.45` and `$34.56`, respectively.

### *`LinearLayout` Id Property Values*

Figure 3.5 shows the **Id**s of the three horizontal `LinearLayout`s in the `GridLayout`'s right column.

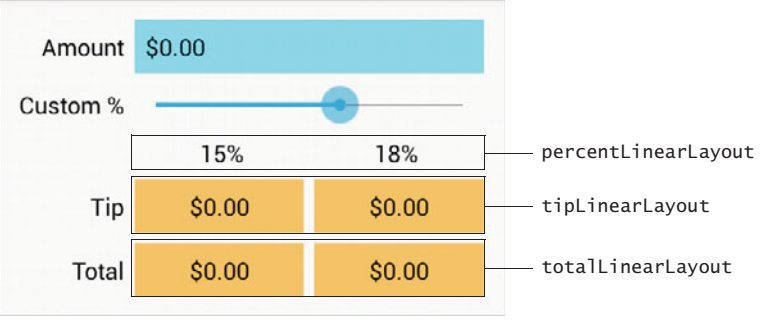



**Fig. 3.5** | **Tip Calculator** GUI's LinearLayouts with their **Id** property values.

### 3.4.2 Creating the `TipCalculator` Project

The Android Developer Tools IDE allows only *one* project with a given name per workspace, so before you create the new project, delete the `TipCalculator` project that you test-drove in Section 3.2. To do so, right click it and select **Delete**. In the dialog that appears, ensure that **Delete project contents on disk** is *not* selected, then click **OK**. This removes the project from the workspace, but leaves the project's folder and files on disk in case you'd like to look at our original app again later.

#### *Creating a New Blank App Project*

Next, create a new **Android Application Project**. Specify the following values in the **New Android Project** dialog's first **New Android Application** step, then press **Next >**:

- **Application Name:** `Tip Calculator`
- **Project Name:** `TipCalculator`
- **Package Name:** `com.deitel.tipcalculator`
- **Minimum Required SDK:** `API18: Android 4.3`
- **Target SDK:** `API19: Android 4.4`
- **Compile With:** `API19: Android 4.4`
- **Theme:** `Holo Light with Dark Action Bar`
- **Create Activity:** `TipCalculator`
- **Build Target:** Ensure that **Android 4.3** is checked

In the **New Android Project** dialog's second **New Android Application** step, leave the default settings, then press **Next >**. In the **Configure Launcher Icon** step, click the **Browse...** button, select the `DeitelGreen.png` app icon image (provided in the `images` folder with the book's examples) and click the **Open** button, then press **Next >**. In the **Create Activity** step, select **Blank Activity** (keep the default activity name), then press **Next >**. In the **Blank Activity** step, leave the default settings, then press **Finish** to create the project. In the **Graphical Layout** editor, select **Nexus 4** from the screen-type drop-down list (as in Fig. 2.12). Once again, we'll use this device as the basis for our design.

### 3.4.3 Changing to a `GridLayout`

The default layout in a **Blank App**'s GUI is a `RelativeLayout`. Here, you'll change that to a `GridLayout`. First, right click the `TextView` in the **Outline** window, then select **Delete** to remove it from the GUI. Next, right click the `RelativeLayout` in the **Outline** window and select **Change Layout...**. In the **Change Layout** dialog, select `GridLayout` and click **OK**. The IDE changes the layout and sets its **Id** to `GridLayout1`. We changed this to `gridLayout` using the **Id** field in the **Properties** window. By default, the `GridLayout`'s **Orientation** property is set to horizontal, indicating that its contents will be laid out row-by-row.

#### *Specifying Two Columns and Default Margins for the `GridLayout`*

Recall that the GUI in Fig. 3.3 consists of two columns. To specify this, select `gridLayout` in the **Outline** window, then change its **Column Count** property to `2` (in the **Properties** window's **GridLayout** group). By default, there are *no margins*—spaces that separate views—

around a GridLayout's cells. Set the GridLayout's **Use Default Margins** property to true to indicate that the GridLayout should place margins around its cells. By default, the GridLayout uses the recommended gap between views (8dp), as specified at

```
http://developer.android.com/design/style/metrics-grids.html
```

### 3.4.4 Adding the TextViews, EditText, SeekBar and LinearLayouts

You'll now build the GUI in Fig. 3.3. You'll start with the basic layout and views in this section. In Section 3.4.5, you'll customize the views' properties to complete the design. As you add each view to the GUI, immediately set its **Id** property using the names in Figs. 3.4–3.5. You can change the selected view's **Id** via the **Properties** window or by right clicking the view (in the **Graphical Layout** editor or **Outline** window), selecting **Edit ID...** and changing the **Id** in the **Rename Resource** dialog that appears.

In the following steps, you'll use the **Outline** window to add views to the GridLayout. When working with layouts, it can be difficult to see the layout's *nested structure* and to place views in the correct locations by dragging them onto the **Graphical Layout** editor window. The **Outline** window makes these tasks easier because it shows the GUI's nested structure. Perform the following steps in the exact order specified—otherwise, the views will *not* appear in the correct order in each row. If this happens, you can reorder views by dragging them in the **Outline** window.

#### *Step 1: Adding Views to the First Row*

The first row consists of the amountTextView in the first column and the amountEditText behind the amountDisplayTextView in the second column. Each time you drop a view or layout onto the gridLayout in the **Outline** window, the view is placed in the layout's *next open cell*, unless you specify otherwise by setting the view's **Row** and **Column** properties. You'll do that in this step so that the amountEditText and amountDisplayTextView are placed in the same cell.

All of the TextViews in this app use the *medium*-sized font from the app's theme. The **Graphical Layout** editor's **Palette** provides *preconfigured* TextViews named **Large**, **Medium** and **Small** (in the **Form Widgets** section) to represent the theme's corresponding text sizes. In each case, the IDE configures the TextView's **Text Appearance** property accordingly. Perform the following tasks to add the two TextViews and the EditText:

1. Drag a **Medium** TextView from the **Palette**'s **Form Widgets** section and drop it on the gridLayout in the **Outline** window. The IDE creates a new TextView named textView1 and nests it in the gridLayout node. The default text "Medium Text" appears in the **Graphical Layout** editor. Change the TextView's **Id** to amountTextView. You'll change its text in Step 6 (Section 3.4.5).
2. This app allows you to enter only *non-negative integers*, which the app divides by 100.0 to display the bill amount. The **Palette**'s **Text Fields** section provides many *preconfigured* EditTexts for various forms of input (e.g., numbers, times, dates, addresses and phone numbers). When the user interacts with an EditText, an appropriate keyboard is displayed based on the EditText's *input type*. When you hover over an EditText in the **Palette**, a *tooltip* indicates the input type. From the **Palette**'s **Text Fields** section, drag a **Number** EditText (displayed with the number **42** on it) and drop it on the gridLayout node in the **Outline** window. Change the

EditText's **Id** to amountEditText. The EditText is placed in the *second* column of the GridLayout's *first* row.

3. Drag another **Medium** TextView onto the gridLayout node in the **Outline** window and change the **Id** to amountDisplayTextView. The new TextView is initially placed in the *first* column of the GridLayout's *second* row. To place it in the *second* column of the GridLayout's *first* row, set this TextView's **Row** and **Column** properties (located in the **Properties** window's **Layout Parameters** section) to the values 0 and 1, respectively.

### *Step 2: Adding Views to the Second Row*

Next, you'll add a TextView and SeekBar to the GridLayout. To do so:

1. Drag a **Medium** TextView (customPercentTextView) from the **Palette**'s **Form Widgets** section onto the gridLayout node in the **Outline** window.
2. Drag a SeekBar (customTipSeekBar) from the **Palette**'s **Form Widgets** section onto the gridLayout node in the **Outline** window.

### *Step 3: Adding Views to the Third Row*

Next, you'll add a LinearLayout containing two TextViews to the GridLayout. To do so:

1. From the **Palette**'s **Layouts** section, drag a **Linear Layout (Horizontal)** (percentLinearLayout) onto the gridLayout node in the **Outline** window.
2. Drag a **Medium** TextView (percent15TextView) onto the percentLinearLayout node in the **Outline** window. This nests the new TextView in the LinearLayout.
3. Drag another **Medium** TextView (percentCustomTextView) onto the percentLinearLayout node in the **Outline** window.
4. The percentLinearLayout and its two nested TextViews should be placed in the second column of the GridLayout. To do so, select the percentLinearLayout in the **Outline** window, then set its **Column** property to 1.

### *Step 4: Adding Views to the Fourth Row*

Next, you'll add a TextView and a LinearLayout containing two more TextViews to the GridLayout. To do so:

1. Drag a **Medium** TextView (tipTextView) onto the gridLayout node.
2. Drag a **Linear Layout (Horizontal)** (tipLinearLayout) onto the gridLayout node.
3. Drag two **Medium** TextViews (tip15TextView and tipCustomTextView) onto the tipLinearLayout node.

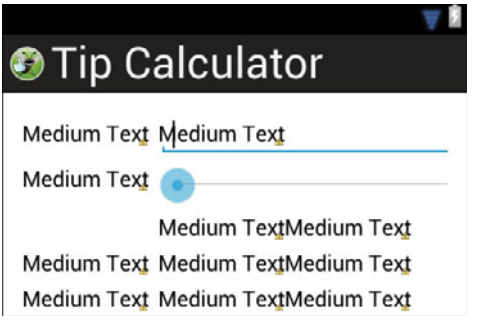
### *Step 5: Adding Views to the Fifth Row*

To create the last row of the GUI, repeat Step 4, using the **Id**s totalTextView, totalLinearLayout, total15TextView and totalCustomTextView.

### *Reviewing the Layout So Far*

The GUI and **Outline** window should now appear as shown in Fig. 3.6. The warning symbols shown in the **Graphical Layout** editor and the **Outline** window will go away as you complete the GUI design in Section 3.4.5.

a) GUI design so far




b) Outline window showing **Tip Calculator** components

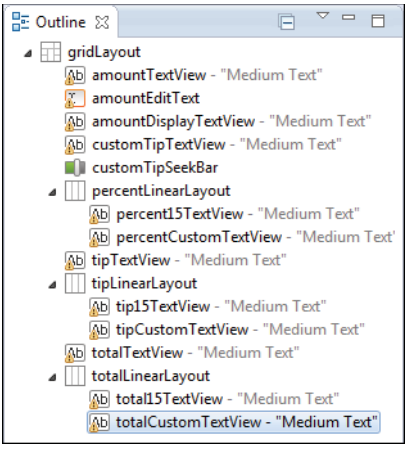



**Fig. 3.6** | The GUI and the IDE's **Outline** window after adding all the views to the `GridLayout`.

### 3.4.5 Customizing the Views to Complete the Design

You'll now complete the app's design by customizing the views' properties and creating several string and dimension resources. As you learned in Section 2.5, literal string values should be placed in the `strings.xml` resource file. Similarly, literal numeric values that specify view dimensions (e.g., widths, heights and spacing) should be placed in the `dimens.xml` resource file.

#### *Step 6: Specifying Literal Text*

Specify the literal text for the `amountTextView`, `customPercentTextView`, `percent15TextView`, `percentCustomTextView`, `tipTextView` and `totalTextView`:

1. Select the `amountTextView` in the **Outline** window.
2. In the **Properties** window, click the ellipsis button next to the **Text** property.
3. In the **Resource Chooser Dialog**, click **New String...**.
4. In the **Create New Android String** dialog, specify `Amount` in the **String** field and `amount` in the **New R.string** field, then click **OK**.
5. In the **Resource Chooser** dialog, click **OK** to set the `amountTextView`'s **Text** property to the string resource identified as `amount`.

Repeat the preceding tasks for the other `TextViews` using the values shown in Fig. 3.7.

| View | String | New R.string |
|---|---|---|
| `customPercentTextView` | `Custom %` | `custom_tip_percentage` |
| `percent15TextView` | `15%` | `fifteen_percent` |

**Fig. 3.7** | String resource values and resource IDs. (Part 1 of 2.)

| View | String | New R.string |
|---|---|---|
| percentCustomTextView | 18% | eighteen_percent |
| tipTextView | Tip | tip |
| totalTextView | Total | total |

**Fig. 3.7** | String resource values and resource IDs. (Part 2 of 2.)

### *Step 7: Right Aligning the `TextView`s in the Left Column*

In Fig. 3.3, each of the left column's `TextView`s is right aligned. For the `amountTextView`, `customPercentTextView`, `tipTextView` and `totalTextView`, set the layout **Gravity** property to `right`—located in the **Layout Parameters** section in the **Properties** window.

### *Step 8: Configuring the `amountTextView`'s Label For Property*

Generally, each `EditText` should have a descriptive `TextView` that helps the user understand the `EditText`'s purpose (also helpful for accessibility)—otherwise, *Android Lint* issues a warning. To fix this, you set the `TextView`'s **Label For** property to the **Id** of the associated `EditText`. Select the `amountTextView` and set its **Label For** property (in the **Properties** window's **View** section) to

```
@+id/amountEditText
```

The `+` is required because the `TextView` is defined *before* the `EditText` in the GUI, so the `EditText` does not yet exist when Android converts the layout's XML into the GUI.

### *Step 9: Configuring the `amountEditText`*

In the final app, the `amountEditText` is *hidden* behind the `amountDisplayTextView` and is configured to allow only *digits* to be entered by the user. Select the `amountEditText` and set the following properties:

1. In the **Properties** window's **Layout Parameters** section, set the **Width** and **Height** to `wrap_content`. This indicates that the `EditText` should be just large enough to fit its content, including any padding.
2. Remove the layout **Gravity** value `fill_horizontal`, leaving the property's value blank. We'll discuss `fill_horizontal` in the next step.
3. Remove the **Ems** property's value, which indicates the `EditText`'s width, measured in uppercase M characters of the view's font. In our `GridLayout`, this causes the second column to be too narrow, so we removed this default setting.
4. In the **Properties** window's **TextView** section, set **Digits** to `0123456789`—this allows *only* digits to be entered, even though the numeric keypad contains minus (-), comma (,), period (.) and space buttons. By default, the **Digits** property is *not* displayed in the **Properties** window, because it's considered to be an advanced property. To display it, click the **Show Advanced Properties** (  ) toggle button at the top of the **Properties** window.
5. We restricted the bill amount to a maximum of *six* digits—so the largest supported bill amount is 9999.99. In the **Properties** window's **TextView** section, set the **Max Length** property to 6.

### *Step 10: Configuring the* `amountDisplayTextView`

To complete the formatting of the `amountDisplayTextView`, select it and set the following properties:

1. In the **Properties** window's **Layout Parameters** section, set the **Width** and **Height** to `wrap_content` to indicate that the `TextView` should be large enough to fit its content.
2. Remove the **Text** property's value—we'll programmatically display text here.
3. In the **Properties** window's **Layout Parameters** section, set the layout **Gravity** to `fill_horizontal`. This indicates that the `TextView` should occupy all remaining horizontal space in this `GridLayout` row.
4. In the **View** section, set the **Background** to `@android:color/holo_blue_bright`. This is one of several *predefined colors* (each starts with `@android:color`) in Android's *Holo* theme. As you start typing the **Background** property's value, a drop-down list of the theme's available colors is displayed. You can also use any *custom color* created from a combination of red, green and blue components called **RGB values**—each is an integer in the range 0–255 that defines the amount of red, green and blue in the color, respectively. Custom colors are defined in *hexadecimal (base 16) format*, so the RGB components are values in the range 00–FF. Android also supports *alpha* (*transparency*) values in the range 0 (*completely transparent*) to 255 (*completely opaque*). To use alpha, you specify the color in the format `#AARRGGBB`, where the first two hexadecimal digits represent the alpha value. If both digits of each color component are the same, you can use the abbreviated formats `#RGB` or `#ARGB`. For example, `#9AC` is treated as `#99AACC` and `#F9AC` is treated as `#FF99AACC`.
5. Finally, you'll add some padding around the `TextView`. To do so, you'll create a new *dimension resource* named `textview_padding`, which you'll use several times in the GUI. A view's **Padding** property specifies space on all sides of the views's content. In the **Properties** window's **View** section, click the **Padding** property's ellipsis button. Click **New Dimension...** to create a new *dimension resource*. Specify `textview_padding` for the **Name** and `8dp` for the **Value** and click **OK**, then select your new *dimension resource* and click **OK**.

### *Step 11: Configuring the* `customPercentTextView`

Notice that the `customPercentTextView` is aligned with the top of the `customTipSeekBar`'s thumb. This looks better if it's *vertically centered*. To do this, in the **Properties** window's **Layout Parameters** section, modify the **Gravity** value from `right` to

```
right|center_vertical
```

The *vertical bar* (`|`) character is used to separate *multiple* **Gravity** values—in this case indicating that the `TextView` should be *right aligned* and *centered vertically* within the grid cell. Also set the `customPercentTextView`'s **Width** and **Height** properties to `wrap_content`.

### *Step 12: Configuring the* `customTipSeekBar`

By default, a `SeekBar`'s range is `0` to `100` and its current value is indicated by its **Progress** property. This app allows custom tip percentages from 0 to 30 and specifies a default of 18. Set the `SeekBar`'s **Max** property to `30` and the **Progress** property to `18`. Also, set the **Width** and **Height** to `wrap_content`.

### *Step 13: Configuring the* `percent15TextView` *and* `percentCustomTextView`

Recall that `GridLayout` does *not* allow you to specify how a view should be sized relative to other views in a given row. This is why we placed the `percent15TextView` and `percentCustomTextView` in a `LinearLayout`, which *does* allow *proportional sizing*. A view's layout **Weight** (in certain layouts, such as `LinearLayout`) specifies the view's relative importance with respect to other views in the layout. By default, all views have a **Weight** of `0`.

In this layout, we set **Weight** to `1` for `percent15TextView` and `percentCustomTextView`—this indicates that they have equal importance, so they should be sized equally. By default, when we added the `percentLinearLayout` to the `GridLayout`, its layout **Gravity** property was set to `fill_horizontal`, so the layout occupies the remaining space in the third row. When the `LinearLayout` is stretched to fill the rest of the row, the `TextViews` each occupy *half* of the `LinearLayout`'s width.

We also wanted each `TextView` to center its text. To do this, in the **Properties** window's **TextView** section, set the **Gravity** property to `center`. This specifies the `TextView`'s text alignment, whereas the *layout* **Gravity** property specifies how a view aligns with respect to the layout.

### *Step 14: Configuring the* `tip15TextView`, `tipCustomTextView`, `total15TextView` *and* `totalCustomTextView`

To finalize these four `TextViews`, perform the following tasks on each:

1. Select the `TextView`.
2. Delete its **Text** value—we'll set this programmatically.
3. Set the **Background** to `@android:color/holo_orange_light`.
4. Set the layout **Gravity** to `center`.
5. Set the layout **Weight** to `1`.
6. Set the layout **Width** to `0dp`—this allows the layout to use the **Weight** to determine the view's width.
7. Set the `TextView` **Gravity** to `center`.
8. Set the `TextView` **Padding** to `@dimen/textview_padding` (the *dimension resource* you created in a previous step).

Notice that there's *no horizontal space* between the `TextViews` in the `tipLinearLayout` and `totalLinearLayout`. To fix this, you'll specify an `8dp` right margin for the `tip15TextView` and `total15TextView`. In the **Properties** window's **Layout Parameters** section, expand the **Margin** section, then set the **Right** margin to `8dp` by creating a new *dimension resource* named `textview_margin`. Next, use this resource to set the `total15TextView`'s **Right** margin.

### *Step 15: Vertically Centering the* `tipTextView` *and* `totalTextView`

To vertically center the `tipTextView` and `totalTextView` with the other views in their respective rows, modify their layout **Gravity** properties from `right` to

```
right|center_vertical
```

When you do this for the `totalTextView`, the `GridLayout` centers this component vertically in the *remaining space from the fifth row to the bottom of the screen*. To fix this problem, drag a **Space** view (in the **Palette**'s **Layout** section) onto the `gridLayout` node in the **Outline**

window. This creates a sixth row that occupies the rest of the screen. As its name implies, a **Space** view occupies space in a GUI. The GUI should now appear as in Fig. 3.8.

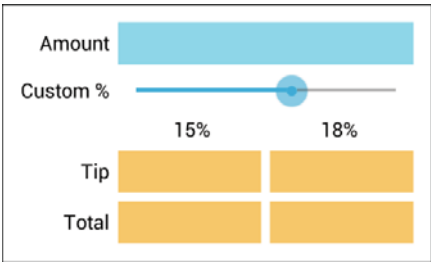



**Fig. 3.8** | Final GUI design.

## 3.5 Adding Functionality to the App

Class `MainActivity` (Figs. 3.9–3.16) implements the **Tip Calculator** app's functionality. It calculates the 15% and custom percentage tips and total bill amounts, and displays them in locale-specific currency format. To view the file, open `src/com.deitel/tipcalculator` and double clck `MainActivity.java`. You'll need to enter most of the code in Figs. 3.9–3.16.

### *The* `package` *and* `import` *Statements*

Figure 3.9 shows the `package` statement and `import` statements in `MainActivity.java`. The `package` statement in line 3 was inserted when you created the project. When you open a Java file in the IDE, the `import` statements are collapsed—one is displayed with a ⊕ to its left. You can click the ⊕ to see the complete list of `import` statements.

```
1  // MainActivity.java
2  // Calculates bills using 15% and custom percentage tips.
3  package com.deitel.tipcalculator;
4
5  import java.text.NumberFormat; // for currency formatting
6
7  import android.app.Activity; // base class for activities
8  import android.os.Bundle; // for saving state information
9  import android.text.Editable; // for EditText event handling
10 import android.text.TextWatcher; // EditText listener
11 import android.widget.EditText; // for bill amount input
12 import android.widget.SeekBar; // for changing custom tip percentage
13 import android.widget.SeekBar.OnSeekBarChangeListener; // SeekBar listener
14 import android.widget.TextView; // for displaying text
15
```

**Fig. 3.9** | `MainActivity`'s `package` and `import` statements.

Lines 5–14 `import` the classes and interfaces the app uses:

- Class `NumberFormat` of package `java.text` (line 5) provides numeric formatting capabilities, such as *locale-specific* currency and percentage formats.

- Class `Activity` of package `android.app` (line 7) provides the basic *lifecycle methods* of an app—we'll discuss these shortly.
- Class `Bundle` of package `android.os` (line 8) represents an app's *state information*. Android gives an app the opportunity to *save its state* before another app appears on the screen. This might occur, for example, when the user *launches another app* or *receives a phone call*. The app that's currently on the screen at a given time is in the *foreground* (the user can interact with it, and the app consumes the CPU) and all other apps are in the *background* (the user cannot interact with them, and they're typically not consuming the CPU). When another app comes into the foreground, the app that was previously in the foreground is given the opportunity to *save its state* as it's sent to the background.
- Interface `Editable` of package `android.text` (line 9) allows you to modify the content and markup of text in a GUI.
- You implement interface `TextWatcher` of package `android.text` (line 10) to respond to events when the user changes the text in an `EditText`.
- Package `android.widget` (lines 11–14) contains the *widgets* (i.e., views) and layouts that are used in Android GUIs. This app uses `EditText` (line 11), `SeekBar` (line 12) and `TextView` (line 14) widgets.
- You implement interface `SeekBar.OnSeekBarChangeListener` of package `android.widget` (line 13) to respond to the user moving the `SeekBar`'s *thumb*.

As you write code with various classes and interfaces, you can use the IDE's **Source > Organize Imports** command to let the IDE insert the `import` statements for you. For cases in which the same class or interface name appears in more than one package, the IDE will let you select the appropriate `import` statement.

### *Tip Calculator App* `Activity` *and the Activity Lifecycle*

Class `MainActivity` (Figs. 3.10–3.16) is the **Tip Calculator** app's `Activity` subclass. When you created the `TipCalculator` project, the IDE generated this class as a subclass of `Activity` and provided an override of class `Activity`'s inherited `onCreate` method (Fig. 3.11). Every `Activity` subclass *must* override this method. The default code for class `MainActivity` also included an `onCreateOptionsMenu` method, which we removed because it's not used in this app. We'll discuss `onCreate` shortly.

```
16  // MainActivity class for the Tip Calculator app
17  public class MainActivity extends Activity
18  {
```

**Fig. 3.10** | Class `MainActivity` is a subclass of `Activity`.

### *Class Variables and Instance Variables*

Lines 20–32 of Fig. 3.11 declare class `MainActivity`'s variables. The `NumberFormat` objects (lines 20–23) are used to format currency values and percentages, respectively. `NumberFormat` `static` method `getCurrencyInstance` returns a `NumberFormat` object that formats values as currency using the device's *default locale*. Similarly, `static` method `getPercentInstance` formats values as percentages using the device's *default locale*.

```
19      // currency and percent formatters
20      private static final NumberFormat currencyFormat =
21         NumberFormat.getCurrencyInstance();
22      private static final NumberFormat percentFormat =
23         NumberFormat.getPercentInstance();
24
25      private double billAmount = 0.0; // bill amount entered by the user
26      private double customPercent = 0.18; // initial custom tip percentage
27      private TextView amountDisplayTextView; // shows formatted bill amount
28      private TextView percentCustomTextView; // shows custom tip percentage
29      private TextView tip15TextView; // shows 15% tip
30      private TextView total15TextView; // shows total with 15% tip
31      private TextView tipCustomTextView; // shows custom tip amount
32      private TextView totalCustomTextView; // shows total with custom tip
33
```

**Fig. 3.11** | `MainActivity` class's instance variables.

The bill amount entered by the user into `amountEditText` will be read and stored as a `double` in `billAmount` (line 25). The custom tip percentage (an integer in the range 0–30) that the user sets by moving the `Seekbar` *thumb* will be multiplied by 0.01 to create a `double` for use in calculations, then stored in `customPercent` (line 26). For example, if you select `25` with the `SeekBar`, `customPercent` will store `0.25`, so the app will multiply the bill amount by `0.25` to calculate the 25% tip.

Line 27 declares the `TextView` that displays the currency-formatted bill amount. Line 28 declares the `TextView` that displays the custom tip percentage based on the `SeekBar` *thumb's* position (see the **18%** in Fig. 3.1(a)). The variables in line 29–32 will refer to the `TextViews` in which the app displays the calculated tips and totals.

### *Overriding Method* `onCreate` *of Class* `Activity`

The `onCreate` method (Fig. 3.12)—which is *auto-generated* with lines 38–39 when you create the app's project—is called by the system when an `Activity` is *started*. Method `onCreate` typically initializes the `Activity`'s instance variables and views. This method should be as simple as possible so that the app *loads quickly*. In fact, if the app takes longer than *five seconds* to load, the operating system will display an **ANR (Application Not Responding) dialog**—giving the user the option to *forcibly terminate the app*. You'll learn how to prevent this problem in Chapter 8.

```
34      // called when the activity is first created
35      @Override
36      protected void onCreate(Bundle savedInstanceState)
37      {
38         super.onCreate(savedInstanceState); // call superclass's version
39         setContentView(R.layout.activity_main); // inflate the GUI
40
```

**Fig. 3.12** | Overriding `Activity` method `onCreate`. (Part 1 of 2.)

```
41          // get references to the TextViews
42          // that MainActivity interacts with programmatically
43          amountDisplayTextView =
44             (TextView) findViewById(R.id.amountDisplayTextView);
45          percentCustomTextView =
46             (TextView) findViewById(R.id.percentCustomTextView);
47          tip15TextView = (TextView) findViewById(R.id.tip15TextView);
48          total15TextView = (TextView) findViewById(R.id.total15TextView);
49          tipCustomTextView = (TextView) findViewById(R.id.tipCustomTextView);
50          totalCustomTextView =
51             (TextView) findViewById(R.id.totalCustomTextView);
52
53          // update GUI based on billAmount and customPercent
54          amountDisplayTextView.setText(
55             currencyFormat.format(billAmount));
56          updateStandard(); // update the 15% tip TextViews
57          updateCustom(); // update the custom tip TextViews
58
59          // set amountEditText's TextWatcher
60          EditText amountEditText =
61             (EditText) findViewById(R.id.amountEditText);
62          amountEditText.addTextChangedListener(amountEditTextWatcher);
63
64          // set customTipSeekBar's OnSeekBarChangeListener
65          SeekBar customTipSeekBar =
66             (SeekBar) findViewById(R.id.customTipSeekBar);
67          customTipSeekBar.setOnSeekBarChangeListener(customSeekBarListener);
68       } // end method onCreate
69
```

**Fig. 3.12** | Overriding `Activity` method `onCreate`. (Part 2 of 2.)

### *`onCreate`'s `Bundle` Parameter*

During the app's execution, the user could change the device's configuration by *rotating the device* or *sliding out a hard keyboard*. For a good experience, the app should continue operating smoothly through such configuration changes. When the system calls `onCreate`, it passes a **`Bundle`** argument containing the `Activity`'s saved state, if any. Typically, you save state in `Activity` methods `onPause` or `onSaveInstanceState` (demonstrated in later apps). Line 38 calls the superclass's `onCreate` method, which is *required* when overriding `onCreate`.

### *Generated `R` Class Contains Resource IDs*

As you build your app's GUI and add *resources* (such as `strings` in the `strings.xml` file or views in the `activity_main.xml` file) to your app, the IDE generates a class named **`R`** that contains *nested classes* representing each type of resource in your project's `res` folder. You can find this class in your project's **gen folder**, which contains generated source-code files. The nested classes are declared `static`, so that you can access them in your code with `R.`*ClassName*. Within class `R`'s nested classes, the IDE creates `static final int` constants that enable you to refer to your app's resources programmatically from your code (as we'll discuss momentarily). Some of the nested classes in class `R` include:

- class **`drawable`**—contains constants for any `drawable` items, such as *images*, that you put in the various `drawable` folders in your app's `res` folder

- class **id**—contains constants for the *views* in your *XML layout files*
- class **layout**—contains constants that represent each *layout file* in your project (such as, `activity_main.xml`)
- class **string**—contains constants for each `String` in the `strings.xml` file.

### *Inflating the GUI*

The call to **setContentView** (line 39) receives the constant **R.layout.activity_main** to indicate which XML file represents `MainActivity`'s GUI—in this case, the constant represents the `main.xml` file. Method `setContentView` uses this constant to load the corresponding XML document, which is then parsed and converted into the app's GUI. This process is known as **inflating** the GUI.

### *Getting References to the Widgets*

Once the layout is *inflated*, you can *get references to the individual widgets* so that you can interact with them programmatically. To do so, you use class `Activity`'s `findViewById` method. This method takes an `int` constant representing a specific view's **Id** and returns a reference to the view. The name of each view's **R.id** constant is determined by the component's **Id** property that you specified when designing the GUI. For example, `amountEditText`'s constant is `R.id.amountEditText`.

Lines 43–51 obtain references to the `TextViews` that are changed by the app. Lines 43–44 obtain a reference to the `amountDisplayTextView` that's updated when the user enters the bill amount. Lines 45–46 obtain a reference to the `percentCustomTextView` that's updated when the user changes the custom tip percentage. Lines 47–51 obtain references to the `TextViews` where the calculated tips and totals are displayed.

### *Displaying Initial Values in the* **TextViews**

Lines 54–55 set `amountDisplayTextView`'s text to the initial `billAmount` (`0.00`) in a *locale-specific* currency format by calling the `currencyFormat` object's **format method**. Next, lines 56–57 call methods `updateStandard` (Fig. 3.13) and `updateCustom` (Fig. 3.14) to display initial values in the tip and total `TextViews`.

### *Registering the Event Listeners*

Lines 60–61 get a reference to the `amountEditText`, and line 62 calls its `addTextChangedListener` method to register the `TextChangedListener` that will respond to *events* generated when the *user changes the text* in the `EditText`. We define this listener (Fig. 3.16) as an *anonymous-inner-class object* that's assigned to the instance variable `amountEditTextWatcher`.

Lines 65–66 get a reference to the `customTipSeekBar` and line 67 calls its `setOnSeekBarChangeListener` method to register the `OnSeekBarChangeListener` that will respond to *events* generated when the user moves the `customTipSeekBar`'s *thumb* to change the custom tip percentage. We define this listener (Fig. 3.15) as an *anonymous-inner-class object* that's assigned to the instance variable `customSeekBarListener`.

### *Method* **updateStandard** *of Class* **MainActivity**

Method `updateStandard` (Fig. 3.13) updates the 15% tip and total `TextViews` each time the user *changes* the bill amount. The method uses the `billAmount` value to calculate the tip amount and the total of the bill amount and tip. Lines 78–79 display the amounts in currency format.

```
70      // updates 15% tip TextViews
71      private void updateStandard()
72      {
73         // calculate 15% tip and total
74         double fifteenPercentTip = billAmount * 0.15;
75         double fifteenPercentTotal = billAmount + fifteenPercentTip;
76
77         // display 15% tip and total formatted as currency
78         tip15TextView.setText(currencyFormat.format(fifteenPercentTip));
79         total15TextView.setText(currencyFormat.format(fifteenPercentTotal));
80      } // end method updateStandard
81
```

**Fig. 3.13** | Method `updateStandard` calculates and displays the 15% tip and total.

### *Method* `updateCustom` *of Class* `MainActivity`

Method `updateCustom` (Fig. 3.14) updates the custom tip and total `TextViews` based on the tip percentage the user selected with the `customTipSeekBar`. Line 86 sets the `percentCustomTextView`'s text to the `customPercent` value formatted as a percentage. Lines 89–90 calculate the `customTip` and `customTotal`. Then, lines 93–94 display the amounts in currency format.

```
82      // updates the custom tip and total TextViews
83      private void updateCustom()
84      {
85         // show customPercent in percentCustomTextView formatted as %
86         percentCustomTextView.setText(percentFormat.format(customPercent));
87
88         // calculate the custom tip and total
89         double customTip = billAmount * customPercent;
90         double customTotal = billAmount + customTip;
91
92         // display custom tip and total formatted as currency
93         tipCustomTextView.setText(currencyFormat.format(customTip));
94         totalCustomTextView.setText(currencyFormat.format(customTotal));
95      } // end method updateCustom
96
```

**Fig. 3.14** | Method `updateCustom` calculates and displays the custom tip and total.

### *Anonymous Inner Class That Implements Interface* `OnSeekBarChangeListener`

Lines 98–120 of Fig. 3.15 create the *anonymous-inner-class* object named `customSeekBarListener` that responds to `customTipSeekBar`'s *events*. If you're not familiar with *anonymous inner classes*, visit the following page:

```
http://bit.ly/AnonymousInnerClasses
```

Line 67 (Fig. 3.12) registered `customSeekBarListener` as `customTipSeekBar`'s `OnSeekBarChangeListener` *event-handling* object. For clarity, we define all but the simplest event-handling objects in this manner so that we do not clutter the `onCreate` method with this code.

```
97      // called when the user changes the position of SeekBar
98      private OnSeekBarChangeListener customSeekBarListener =
99         new OnSeekBarChangeListener()
100        {
101           // update customPercent, then call updateCustom
102           @Override
103           public void onProgressChanged(SeekBar seekBar, int progress,
104              boolean fromUser)
105           {
106              // sets customPercent to position of the SeekBar's thumb
107              customPercent = progress / 100.0;
108              updateCustom(); // update the custom tip TextViews
109           } // end method onProgressChanged
110
111           @Override
112           public void onStartTrackingTouch(SeekBar seekBar)
113           {
114           } // end method onStartTrackingTouch
115
116           @Override
117           public void onStopTrackingTouch(SeekBar seekBar)
118           {
119           } // end method onStopTrackingTouch
120        }; // end OnSeekBarChangeListener
121
```

**Fig. 3.15** | Anonymous inner class that implements interface `OnSeekBarChangeListener` to respond to the events of the `customSeekBar`.

*Overriding Method* **`onProgressChanged`** *of Interface* **`OnSeekBarChangeListener`**
Lines 102–119 implement interface `OnSeekBarChangeListener`'s methods. Method `onProgressChanged` is called whenever the `SeekBar`'s *thumb* position *changes*. Line 107 calculates `customPercent` using the method's `progress` parameter—an `int` representing the `SeekBar`'s *thumb* position. We divide this by 100.0 to get the custom percentage. Line 108 calls method `updateCustom` to recalculate and display the custom tip and total.

*Overriding Methods* **`onStartTrackingTouch`** *and* **`onStopTrackingTouch`** *of Interface* **`OnSeekBarChangeListener`**
Java requires that you override *every* method in an *interface* that you *implement*. This app does *not* need to know when the user *starts* moving the slider's thumb (`onStartTrackingTouch`) or *stops* moving it (`onStopTrackingTouch`), so we simply provide an *empty* body for each (lines 111–119) to *fulfill* the *interface contract*.

*Anonymous Inner Class That Implements Interface* **`TextWatcher`**
Lines 123–156 of Fig. 3.16 create the *anonymous-inner-class* object `amountEditTextWatcher` that responds to `amountEditText`'s *events*. Line 62 registered this object to *listen* for `amountEditText`'s events that occur when the text changes.

*Overriding Method* **`onTextChanged`** *of Interface* **`TextWatcher`**
The `onTextChanged` method (lines 126–144) is called whenever the text in the `amountEditText` is *modified*. The method receives four parameters. In this example, we use only

```
122    // event-handling object that responds to amountEditText's events
123    private TextWatcher amountEditTextWatcher = new TextWatcher()
124    {
125       // called when the user enters a number
126       @Override
127       public void onTextChanged(CharSequence s, int start,
128          int before, int count)
129       {
130          // convert amountEditText's text to a double
131          try
132          {
133             billAmount = Double.parseDouble(s.toString()) / 100.0;
134          } // end try
135          catch (NumberFormatException e)
136          {
137             billAmount = 0.0; // default if an exception occurs
138          } // end catch
139
140          // display currency formatted bill amount
141          amountDisplayTextView.setText(currencyFormat.format(billAmount));
142          updateStandard(); // update the 15% tip TextViews
143          updateCustom(); // update the custom tip TextViews
144       } // end method onTextChanged
145
146       @Override
147       public void afterTextChanged(Editable s)
148       {
149       } // end method afterTextChanged
150
151       @Override
152       public void beforeTextChanged(CharSequence s, int start, int count,
153          int after)
154       {
155       } // end method beforeTextChanged
156    }; // end amountEditTextWatcher
157 } // end class MainActivity
```

**Fig. 3.16** | Anonymous inner class that implements interface `TextWatcher` to respond to the events of the `amountEditText`.

`CharSequence s`, which contains a copy of `amountEditText`'s text. The other parameters indicate that the `count` characters starting at `start` *replaced* previous text of length `before`.

Line 133 converts the user input from `amountEditText` to a `double`. We allow users to enter only whole numbers in pennies, so we divide the converted value by 100.0 to get the actual bill amount—e.g., if the user enters 2495, the bill amount is 24.95. Lines 142–143 call `updateStandard` and `updateCustom` to recalculate and display the tips and totals.

### *Other Methods of the* `amountEditTextWatcher` `TextWatcher`

This app does *not* need to know what changes are about to be made to the text (`beforeTextChanged`) or that the text has already been changed (`afterTextChanged`), so we simply override each of these `TextWatcher` interface methods with an *empty* body (lines 146–155) to *fulfill the interface contract*.

## 3.6 AndroidManifest.xml

In this section, you'll modify the `AndroidManifest.xml` file to specify that this app's `Activity` supports only a device's *portrait* orientation and that the *soft keypad* should *always* remain on the screen. You'll use the IDE's **Android Manifest** editor to specify these settings. To open the **Android Manifest** editor, double click the app's `AndroidManifest.xml` file in the **Package Explorer**. At the bottom of the editor, click the **Application** tab (Fig. 3.17), then select the `MainActivity` node in the **Application Nodes** section at the bottom of the window. This displays settings for the `MainActivity` in the **Attributes for com.deitel.tipcalculator.MainActivity** section.

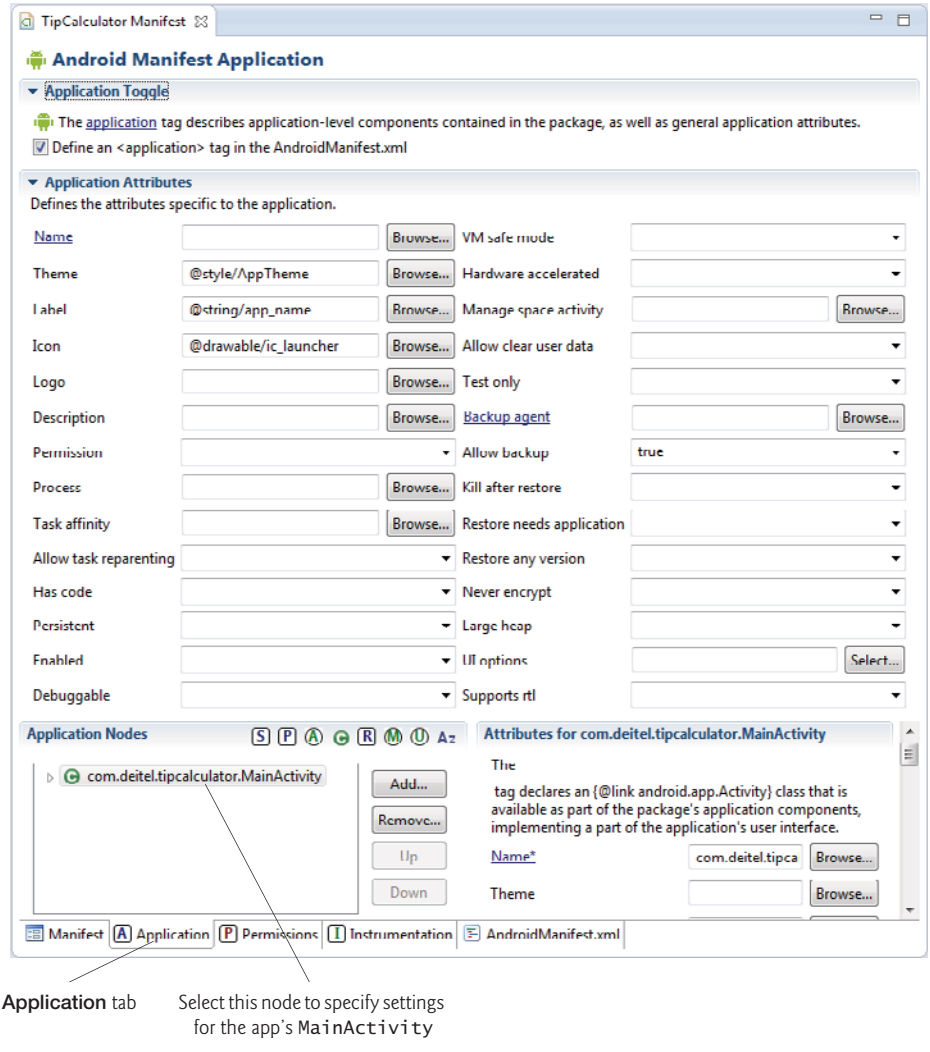



**Fig. 3.17** | **Android Manifest** editor's **Application** tab.

### *Configuring `MainActivity` for Portrait Orientation*

In general, most apps should support *both* portrait and landscape orientations. In *portrait* orientation, the device's height is greater than its width. In *landscape orientation*, the device's width is greater than its height. In the **Tip Calculator** app, rotating the device to landscape orientation on a typical phone would cause the numeric keypad to obscure most of the **Tip Calculator**'s GUI. For this reason, you'll configure `MainActivity` to support *only* portrait orientation. In the **Android Manifest** editor's **Attributes for com.deitel.tipcalculator.MainActivity** section, scroll down to the **Screen orientation** option and select `portrait`.

### *Forcing the Soft Keypad to Always Display for `MainActivity`*

In the **Tip Calculator** app, the soft keypad should be displayed immediately when the app executes and should remain on the screen at all times. In the **Android Manifest** editor's **Attributes for com.deitel.tipcalculator.MainActivity** section, scroll down to the **Window soft input mode** option and select `stateAlwaysVisible`. Note that this will *not* display the soft keyboard if a hard keyboard is present.

## 3.7 Wrap-Up

In this chapter, you created your first *interactive* Android app—the **Tip Calculator**. We overviewed the app's capabilities, then you test-drove it to calculate standard and custom tips based on the bill amount entered. You followed detailed step-by-step instructions to build the app's GUI using the Android Developer Tools IDE's **Graphical Layout** editor, **Outline** window and **Properties** window. We also walked through the code of the `Activity` subclass `MainActivity`, which defined the app's functionality.

In the app's GUI, you used a `GridLayout` to arrange the views into rows and columns. You displayed text in `TextViews` and received input from an `EditText` and a `SeekBar`.

The `MainActivity` class required many Java object-oriented programming capabilities, including classes, objects, methods, interfaces, anonymous inner classes and inheritance. We explained the notion of inflating the GUI from its XML file into its screen representation. You learned about Android's `Activity` class and part of the `Activity` lifecycle. In particular, you overrode the `onCreate` method to initialize the app when it's launched. In the `onCreate` method, you used `Activity` method `findViewById` to get references to each of the views that the app interacts with programmatically. You defined an anonymous inner class that implements the `TextWatcher` interface so the app can calculate new tips and totals as the user changes the text in the `EditText`. You also defined an anonymous inner class that implements the `OnSeekBarChangeListener` interface so the app can calculate a new custom tip and total as the user changes the custom tip percentage by moving the `SeekBar`'s thumb.

Finally, you opened the `AndroidManifest.xml` file in the IDE's **Android Manifest** editor to specify that the `MainActivity` supports only portrait orientation and that the `MainActivity` should always display the keypad.

Using the IDE's **Graphical Layout** editor, **Outline** window, **Properties** window and **Android Manifest** editor enabled you to build this app without manipulating the XML in the project's resource files and `AndroidManifest.xml` file.

In the next chapter, we introduce collections while building the **Twitter® Searches** app. Many mobile apps display lists of items. You'll do this by using a `ListActivity` containing a `ListView` that's bound to an `ArrayList<String>`. You'll also store app data as user preferences and learn how to launch the device's web browser to display a web page.

# Index

## B



## C

## E



## F

## G

## J



## K



## L

## M



## N



## O

## P

## Q



## R

## S