



Lauren Darcey  
Shane Conder

Third Edition  
**Features Android 4.0**

# Android<sup>TM</sup>

## Wireless Application Development

Volume I: Android Essentials

**Developer's Library**



# Android™ Wireless Application Development

---

Volume 1: Android Essentials

Third Edition

*This page intentionally left blank*

# Android™ Wireless Application Development

---

Volume 1: Android Essentials

Third Edition

Lauren Darcey  
Shane Conder

 Addison-Wesley

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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

**U.S. Corporate and Government Sales**  
**1-800-382-3419**  
**[corpsales@pearsontechgroup.com](mailto:corpsales@pearsontechgroup.com)**

For sales outside of the U.S., please contact

**International Sales**  
**[international@pearsoned.com](mailto:international@pearsoned.com)**

Visit us on the Web: [informit.com/aw](http://informit.com/aw)

Library of Congress Cataloging-in-Publication Data:  
Darcey, Lauren, 1977-

Android wireless application development / Lauren Darcey, Shane Conder.  
v. cm.

Includes bibliographical references and index.

Contents: v. 1. Android essentials

ISBN 978-0-321-81383-1 (pbk. : alk. paper)

1. Application software—Development. 2. Android (Electronic resource) 3. Mobile computing. I. Conder, Shane, 1975-. II. Title.

QA76.76.A65D258 2012b

005.1-dc23

2011049390

Copyright © 2012 Lauren Darcey and Shane Conder

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

Android is the trademark of Google, Inc. Pearson Education does not assert any right to the use of the Android trademark and neither Google nor any other third party having any claim in the Android trademark have sponsored or are affiliated with the creation and development of this book.

Some figures that appear in this book have been reproduced from or are modifications based on work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License (<http://creativecommons.org/licenses/by/2.5/>).

ISBN-13: 978-0-321-81383-1

ISBN-10: 0-321-81383-9

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

First Printing: February 2012

<b>Editor-in-Chief</b>	Mark Taub
<b>Acquisitions Editor</b>	Trina MacDonald
<b>Development Editor</b>	Songlin Qiu
<b>Managing Editor</b>	Kristy Hart
<b>Project Editor</b>	Lori Lyons
<b>Copy Editor</b>	Bart Reed
<b>Indexer</b>	Heather McNeill
<b>Proofreader</b>	Paula Lowell
<b>Technical Reviewers</b>	Douglas Jones Mike Wallace Mark Gjoel
<b>Publishing Coordinator</b>	Olivia Basegio
<b>Multimedia Developer</b>	Dan Scherf
<b>Book Designer</b>	Gary Adair
<b>Compositor</b>	Nonie Ratcliff



*This book is dedicated to Chickpea.*



# **Contents at a Glance**

Introduction **1**

## **I: An Overview of the Android Platform**

- 1** Introducing Android **11**
- 2** Setting Up Your Android Development Environment **37**
- 3** Writing Your First Android Application **53**
- 4** Mastering the Android Development Tools **83**

## **II: Android Application Basics**

- 5** Understanding the Anatomy of an Android Application **103**
- 6** Defining Your Application Using the Android Manifest File **119**
- 7** Managing Application Resources **137**

## **III: Android User Interface Design Essentials**

- 8** Exploring User Interface Screen Elements **171**
- 9** Designing User Interfaces with Layouts **199**
- 10** Working with Fragments **233**
- 11** Working with Dialogs **251**

## **IV: Android Application Design Essentials**

- 12** Using Android Preferences **263**
- 13** Working with Files and Directories **275**
- 14** Using Content Providers **285**
- 15** Designing Compatible Applications **301**

## **V: Publishing and Distributing Android Applications**

- 16 The Android Software Development Process 325**
- 17 Designing and Developing Bulletproof Android Applications 347**
- 18 Testing Android Applications 363**
- 19 Publishing Your Android Application 377**

## **VI: Appendixes**

- A The Android Emulator Quick-Start Guide 399**
- B The Android DDMS Quick-Start Guide 423**
- C Eclipse IDE Tips and Tricks 439**
- Index 449**

# Table of Contents

## **Introduction 1**

Who Should Read This Book	1
Key Questions Answered in This Volume	2
How These Books Are Structured	2
An Overview of Changes in This Edition	4
Development Environment Used in This Book	5
Supplementary Materials Available	6
Where to Find More Information	6
Conventions Used in This Book	7
Contacting the Authors	8

## **I: An Overview of the Android Platform**

### **1 Introducing Android 11**

A Brief History of Mobile Software Development	11
Way Back When	11
“The Brick”	13
Wireless Application Protocol (WAP)	15
Proprietary Mobile Platforms	17
The Open Handset Alliance	19
Google Goes Wireless	19
Forming the Open Handset Alliance	19
Manufacturers: Designing Android Devices	20
Mobile Operators: Delivering the Android Experience	21
Apps Drive Device Sales: Developing Android Applications	22
Taking Advantage of All Android Has to Offer	22
The Android Marketplace: Where We’re at Now	22
Android Platform Differences	23
Android: A Next-Generation Platform	24
Free and Open Source	25
Familiar and Inexpensive Development Tools	25
Reasonable Learning Curve for Developers	26

Enabling Development of Powerful Applications	26
Rich, Secure Application Integration	26
No Costly Obstacles to Publication	27
A “Free Market” for Applications	27
A Growing Platform	28
The Android Platform	29
Android’s Underlying Architecture	29
Security and Permissions	31
Developing Android Applications	32
Summary	35
References and More Information	35
<b>2 Setting Up Your Android Development Environment</b>	<b>37</b>
Configuring Your Development Environment	37
Configuring Your Operating System for Device Debugging	39
Configuring Your Android Hardware for Debugging	39
Upgrading the Android SDK	41
Problems with the Android Software Development Kit	41
Exploring the Android SDK	42
Understanding the Android SDK License Agreement	42
Reading the Android SDK Documentation	43
Exploring the Core Android Application Framework	43
Exploring the Core Android Tools	46
Exploring the Android Sample Applications	50
Summary	52
References and More Information	52
<b>3 Writing Your First Android Application</b>	<b>53</b>
Testing Your Development Environment	53
Adding the Snake Project to Your Eclipse Workspace	54
Creating an Android Virtual Device (AVD) for Your Snake Project	56

Creating a Launch Configuration for Your Snake Project	58
Running the Snake Application in the Android Emulator	59
Building Your First Android Application	62
Creating and Configuring a New Android Project	62
Core Files and Directories of the Android Application	65
Creating an AVD for Your Project	65
Creating a Launch Configuration for Your Project	66
Running Your Android Application in the Emulator	67
Debugging Your Android Application in the Emulator	69
Adding Logging Support to Your Android Application	73
Adding Some Media Support to Your Application	74
Debugging Your Application on the Hardware	78
Summary	80
References and More Information	81

#### **4 Mastering the Android Development Tools 83**

Using the Android Documentation	83
Leveraging the Android Emulator	85
Viewing Application Log Data with LogCat	86
Debugging Applications with DDMS	87
Using Android Debug Bridge (ADB)	87
Using the Resource Editors and UI Designer	88
Using the Android Hierarchy Viewer	91
Launching the Hierarchy Viewer	92
Working in Layout View Mode	92
Optimizing Your User Interface	94
Working in Pixel Perfect Mode	94
Working with Nine-Patch Stretchable Graphics	95
Working with Other Android Tools	98
Summary	99
References and More Information	100

## II: Android Application Basics

### 5 Understanding the Anatomy of an Android Application 103

Mastering Important Android Terminology 103

Using the Application Context 104

    Retrieving the Application Context 104

    Using the Application Context 104

Performing Application Tasks with Activities 106

    The Lifecycle of an Android Activity 106

Organizing Activity Components with Fragments 111

Managing Activity Transitions with Intents 113

    Transitioning Between Activities with Intents 113

    Organizing Application Navigation with Activities and Intents 115

Working with Services 116

Receiving and Broadcasting Intents 117

Summary 117

    References and More Information 118

### 6 Defining Your Application Using the Android Manifest File 119

Configuring Android Applications Using the Android Manifest File 119

    Editing the Android Manifest File 120

    Managing Your Application's Identity 124

    Versioning Your Application 125

    Setting the Application Name and Icon 125

    Enforcing Application System Requirements 125

    Targeting Specific SDK Versions 126

    Enforcing Application Platform Requirements 129

    Working with External Libraries 130

    Other Application Configuration Settings and Filters 131

    Registering Activities in the Android Manifest 131

    Designating a Primary Entry Point Activity for Your Application Using an Intent Filter 132

Configuring Other Intent Filters	132
Registering Other Application Components	133
Working with Permissions	133
Registering Permissions Your Application Requires	133
Registering Permissions Your Application Enforces	134
Exploring Other Manifest File Settings	135
Summary	136
References and More Information	136
<b>7 Managing Application Resources</b>	<b>137</b>
What Are Resources?	137
Storing Application Resources	137
Resource Value Types	138
Accessing Resources Programmatically	142
Setting Simple Resource Values Using Eclipse	143
Working with Different Types of Resources	146
Working with String Resources	146
Using String Resources as Format Strings	147
Working with String Arrays	149
Working with Boolean Resources	149
Working with Integer Resources	150
Working with Colors	151
Working with Dimensions	152
Working with Simple Drawables	153
Working with Images	154
Working with Animation	156
Working with Menus	158
Working with XML Files	159
Working with Raw Files	160
References to Resources	161
Working with Layouts	162
Referencing System Resources	167
Summary	168
References and More Information	168

### III: Android User Interface Design Essentials

#### 8 Exploring User Interface Screen Elements 171

Introducing Android Views and Layouts	171
Introducing the Android View	171
Introducing the Android Controls	171
Introducing the Android Layout	172
Displaying Text to Users with TextView	173
Configuring Layout and Sizing	173
Creating Contextual Links in Text	174
Retrieving Data from Users with EditText	176
Retrieving Text Input Using EditText Controls	176
Constraining User Input with Input Filters	178
Helping the User with Autocompletion	179
Giving Users Choices Using Spinner Controls	181
Allowing Simple User Selections with Buttons, Check Boxes, Switches, and Radio Groups	183
Using Basic Buttons	184
Using CheckBox and ToggleButton Controls	186
Using RadioGroup and RadioButton	187
Retrieving Dates and Times from Users	190
Using Indicators to Display Data to Users	191
Indicating Progress with ProgressBar	192
Adjusting Progress with SeekBar	194
Displaying Rating Data with RatingBar	194
Showing Time Passage with the Chronometer	195
Displaying the Time	196
Summary	197
References and More Information	198

#### 9 Designing User Interfaces with Layouts 199

Creating User Interfaces in Android	199
Creating Layouts Using XML Resources	199
Creating Layouts Programmatically	201
Organizing Your User Interface	203
Using ViewGroup Subclasses for Layout Design	204
Using ViewGroup Subclasses as View Containers	204

Using Built-in Layout Classes	205
Using FrameLayout	207
Using LinearLayout	209
Using RelativeLayout	211
Using TableLayout	214
Using GridLayout	216
Using Multiple Layouts on a Screen	220
Using Container Control Classes	220
Using Data-Driven Containers	221
Organizing Screens with Tabs	226
Adding Scrolling Support	229
Exploring Other View Containers	230
Summary	231
References and More Information	231
<b>10 Working with Fragments</b>	<b>233</b>
Understanding Fragments	233
Understanding the Fragment Lifecycle	234
Working with Special Types of Fragments	237
Designing Fragment-Based Applications	238
Using the Android Support Package	247
Adding Fragment Support to Legacy Applications	247
Using Fragments in New Applications Targeting Older Platforms	248
Linking the Android Support Package to Your Project	248
Summary	249
References and More Information	250
<b>11 Working with Dialogs</b>	<b>251</b>
Choosing Your Dialog Implementation	251
Exploring the Different Types of Dialogs	252
Working with Dialogs: The Legacy Method	253
Tracing the Lifecycle of a Dialog	254
Working with Custom Dialogs	256
Working with Dialogs: The Fragment Method	257
Summary	260
References and More Information	260

## IV: Android Application Design Essentials

### 12 Using Android Preferences 263

Working with Application Preferences	263
Determining When Preferences Are Appropriate	263
Storing Different Types of Preference Values	264
Creating Private Preferences for Use by a Single Activity	264
Creating Shared Preferences for Use by Multiple Activities	265
Searching and Reading Preferences	265
Adding, Updating, and Deleting Preferences	266
Reacting to Preference Changes	267
Finding Preferences Data on the Android File System	267
Creating Manageable User Preferences	268
Creating a Preference Resource File	269
Using the PreferenceActivity Class	270
Summary	273
References and More Information	273

### 13 Working with Files and Directories 275

Working with Application Data on the Device	275
Practicing Good File Management	276
Understanding Android File Permissions	277
Working with Files and Directories	277
Exploring with the Android Application Directories	278
Working with Other Directories and Files on the Android File System	282
Summary	284
References and More Information	284

### 14 Using Content Providers 285

Exploring Android's Content Providers	285
Using the MediaStore Content Provider	286
Using the CallLog Content Provider	288
Using the Browser Content Provider	289

Using the CalendarContract Content Provider	291
Using the UserDictionary Content Provider	291
Using the VoicemailContract Content Provider	291
Using the Settings Content Provider	292
Using the Contacts Content Providers	292
Modifying Content Providers Data	297
Adding Records	297
Updating Records	298
Deleting Records	298
Using Third-Party Content Providers	299
Summary	300
References and More Information	300
<b>15 Designing Compatible Applications</b>	<b>301</b>
Maximizing Application Compatibility	301
Designing User Interfaces for Compatibility	303
Working with Fragments	305
Leveraging the Android Support Package	305
Supporting Specific Screen Types	305
Working with Nine-Patch Stretchable Graphics	306
Using the Working Square Principle	306
Providing Alternative Application Resources	308
Understanding How Resources Are Resolved	308
Organizing Alternative Resources with Qualifiers	309
Providing Resources for Different Orientations	316
Using Alternative Resources Programmatically	316
Organizing Application Resources Efficiently	316
Targeting Tablets, TVs, and Other New Devices	318
Targeting Tablet Devices	318
Targeting Google TV Devices	319
Summary	321
References and More Information	321
<b>V: Publishing and Distributing Android Applications</b>	
<b>16 The Android Software Development Process</b>	<b>325</b>
An Overview of the Mobile Development Process	325
Choosing a Software Methodology	326

Understanding the Dangers of Waterfall Approaches	326
Understanding the Value of Iteration	327
Gathering Application Requirements	327
Determining Project Requirements	327
Developing Use Cases for Mobile Applications	329
Incorporating Third-Party Requirements	330
Managing a Device Database	330
Assessing Project Risks	333
Identifying Target Devices	333
Acquiring Target Devices	335
Determining the Feasibility of Application Requirements	336
Understanding Quality Assurance Risks	336
Writing Essential Project Documentation	337
Developing Test Plans for Quality Assurance Purposes	338
Providing Documentation Required by Third Parties	338
Providing Documentation for Maintenance and Porting	338
Leveraging Configuration Management Systems	339
Choosing a Source Control System	339
Implementing an Application Version System That Works	339
Designing Mobile Applications	340
Understanding Mobile Device Limitations	340
Exploring Common Mobile Application Architectures	340
Designing for Extensibility and Maintenance	341
Designing for Application Interoperability	342
Developing Mobile Applications	342
Testing Mobile Applications	343
Deploying Mobile Applications	343
Determining Target Markets	344
Supporting and Maintaining Mobile Applications	344
Track and Address Crashes Reported by Users	345
Testing Firmware Upgrades	345

Maintaining Adequate Application Documentation	345
Managing Live Server Changes	345
Identifying Low-Risk Porting Opportunities	345
Summary	346
References and More Information	346

## **17 Designing and Developing Bulletproof Android Applications 347**

Best Practices in Designing Bulletproof Mobile Applications	347
Meeting Mobile Users' Demands	348
Designing User Interfaces for Mobile Devices	348
Designing Stable and Responsive Mobile Applications	349
Designing Secure Mobile Applications	351
Designing Mobile Applications for Maximum Profit	351
Leveraging Third-Party Quality Standards	352
Designing Mobile Applications for Ease of Maintenance and Upgrades	353
Leveraging Android Tools for Application Design	354
Avoiding Silly Mistakes in Android Application Design	355
Best Practices in Developing Bulletproof Mobile Applications	355
Designing a Development Process That Works for Mobile Development	356
Testing the Feasibility of Your Application Early and Often	356
Using Coding Standards, Reviews, and Unit Tests to Improve Code Quality	357
Handling Defects Occurring on a Single Device	359
Leveraging Android Tools for Development	360
Avoiding Silly Mistakes in Android Application Development	360
Summary	361
References and More Information	361

**18 Testing Android Applications 363**

Best Practices in Testing Mobile Applications	363
Designing a Mobile Application Defect Tracking System	363
Managing the Testing Environment	365
Maximizing Testing Coverage	367
Leveraging Android Tools for Android Application Testing	374
Avoiding Silly Mistakes in Android Application Testing	375
Summary	376
References and More Information	376

**19 Publishing Your Android Application 377**

Choosing the Right Distribution Model	377
Protecting Your Intellectual Property	378
Billing the User	379
Packaging Your Application for Publication	380
Preparing Your Code for Packaging	380
Packing and Signing Your Application	382
Testing the Release Version of Your Application Package	384
Distributing Your Application	385
Publishing on the Android Market	385
Signing Up for a Developer Account on the Android Market	385
Uploading Your Application to the Android Market	387
Uploading Application Marketing Assets	388
Configuring Application Listing Details	388
Configuring Application Publishing Options	390
Configuring Application Contact and Consent Information	390
Publishing Your Application on the Android Market	392
Managing Your Application on the Android Market	392
Publishing Using Other Alternatives	393
Self-Publishing Your Application	394

Summary	395
References and More Information	395

## VI: Appendixes

### A The Android Emulator Quick-Start Guide 399

Simulating Reality: The Emulator's Purpose	399
Working with Android Virtual Devices (AVDs)	401
Using the Android Virtual Device Manager	402
Creating an AVD	403
Launching the Emulator with a Specific AVD	407
Maintaining Emulator Performance	407
Configuring Emulator Startup Options	408
Launching an Emulator to Run an Application	408
Launching an Emulator from the Android Virtual Device Manager	410
Configuring the GPS Location of the Emulator	411
Calling Between Two Emulator Instances	413
Messaging between Two Emulator Instances	415
Interacting with the Emulator through the Console	416
Using the Console to Simulate Incoming Calls	416
Using the Console to Simulate SMS Messages	416
Using the Console to Send GPS Coordinates	418
Using the Console to Monitor Network Status	418
Using the Console to Manipulate Power Settings	418
Using Other Console Commands	419
Enjoying the Emulator	419
Understanding Emulator Limitations	420
References and More Information	421

### B The Android DDMS Quick-Start Guide 423

Using DDMS with Eclipse and as a Standalone Application	423
Getting Up to Speed Using Key Features of DDMS	424
Working with Processes, Threads, and the Heap	425
Attaching a Debugger to an Android Application	425
Stopping a Process	426

Monitoring Thread Activity of an Android Application	426
Monitoring Heap Activity	427
Prompting Garbage Collection	428
Creating and Using an HPROF File	429
Using the Allocation Tracker	430
Working with the File Explorer	430
Browsing the File System of an Emulator or Device	432
Copying Files from the Emulator or Device	432
Copying Files to the Emulator or Device	433
Deleting Files on the Emulator or Device	433
Working with the Emulator Control	434
Simulating Incoming Voice Calls	434
Simulating Incoming SMS Messages	434
Sending a Location Fix	435
Taking Screen Captures of the Emulator and Device Screens	435
Working with Application Logging	436
<b>C Eclipse IDE Tips and Tricks 439</b>	
Organizing Your Eclipse Workspace	439
Integrating with Source Control Services	439
Repositioning Tabs within Perspectives	440
Maximizing Windows	440
Minimizing Windows	440
Viewing Windows, Side by Side	440
Viewing Two Sections of the Same File	441
Closing Unwanted Tabs	441
Keeping Windows Under Control	441
Creating Custom Log Filters	441
Searching Your Project	442
Organizing Eclipse Tasks	442
Writing Code in Java	443
Using Autocomplete	443
Creating New Classes and Methods	443
Organizing Imports	443

Formatting Code	444
Renaming Almost Anything	444
Refactoring Code	444
Reorganizing Code	446
Using QuickFix	446
Providing Javadoc-Style Documentation	446
Resolving Mysterious Build Errors	447

**Index 449**

## Acknowledgments

This book would never have been written without the guidance and encouragement we received from a number of supportive individuals, including our editorial team, coworkers, friends, and family. We'd like to thank the Android developer community, Google, and the Open Handset Alliance for their vision and expertise. Throughout this project, our editorial team at Pearson Education (Addison-Wesley) always had the right mix of professionalism and encouragement. Thanks especially to Trina MacDonald, Olivia Basegio, Songlin Qiu, and our crack team of technical reviewers: Doug Jones, Mike Wallace, and Mark Gjoel, (as well as Dan Galpin, Tony Hillerson, Ronan Schwarz, and Charles Stearns, who reviewed previous editions). Dan Galpin also graciously provided the clever Android graphics used for Tips, Notes, and Warnings. We'd also like to thank Ray Rischpater for his longtime encouragement and advice on technical writing. Amy Badger must be commended for her wonderful waterfall illustration, and we also thank Hans Bodlaender for letting us use the nifty chess font he developed as a hobby project.

## About the Authors

**Lauren Darcey** is responsible for the technical leadership and direction of a small software company specializing in mobile technologies, including Android, iOS, Blackberry, Palm Pre, BREW, and J2ME and consulting services. With more than two decades of experience in professional software production, Lauren is a recognized authority in application architecture and the development of commercial-grade mobile applications. Lauren received a B.S. in Computer Science from the University of California, Santa Cruz.

She spends her copious free time traveling the world with her geeky mobile-minded husband and is an avid nature photographer. Her work has been published in books and newspapers around the world. In South Africa, she dove with 4-meter-long great white sharks and got stuck between a herd of rampaging hippopotami and an irritated bull elephant. She's been attacked by monkeys in Japan, gotten stuck in a ravine with two hungry lions in Kenya, gotten thirsty in Egypt, narrowly avoided a coup d'état in Thailand, geocached her way through the Swiss Alps, drank her way through the beer halls of Germany, slept in the crumbling castles of Europe, and gotten her tongue stuck to an iceberg in Iceland (while being watched by a herd of suspicious wild reindeer).

**Shane Conder** has extensive development experience and has focused his attention on mobile and embedded development for the past decade. He has designed and developed many commercial applications for Android, iOS, BREW, Blackberry, J2ME, Palm, and Windows Mobile—some of which have been installed on millions of phones worldwide. Shane has written extensively about the mobile industry and evaluated mobile development platforms on his tech blogs and is well-known within the blogosphere. Shane received a B.S. in Computer Science from the University of California.

A self-admitted gadget freak, Shane always has the latest smartphone, tablet, or other mobile device. He can often be found fiddling with the latest technologies, such as cloud services and mobile platforms, and other exciting, state-of-the-art technologies that activate the creative part of his brain. He also enjoys traveling the world with his geeky wife, even if she did make him dive with 4-meter-long great white sharks and almost get eaten by a lion in Kenya. He admits that he has to take at least two phones with him when backpacking—even though there is no coverage—and that he snickered and whipped out his Android phone to take a picture when Laurie got her tongue stuck to that iceberg in Iceland, and that he is catching on that he should be writing his own bio.

# Introduction

Pioneered by the Open Handset Alliance and Google, Android is a popular, free, open-source mobile platform that has taken the wireless world by storm. This book, and the next volume, *Android Wireless Application Development Volume II: Advanced Topics*, provide comprehensive guidance for software development teams on designing, developing, testing, debugging, and distributing professional Android applications. If you're a veteran mobile developer, you can find tips and tricks to streamline the development process and take advantage of Android's unique features. If you're new to mobile development, these books provide everything you need to make a smooth transition from traditional software development to mobile development—specifically, its most promising platform: Android.

## Who Should Read This Book

This book include tips for successful mobile development based upon our years in the mobile industry and covers everything you need to know in order to run a successful Android project from concept to completion. We cover how the mobile software process differs from traditional software development, including tricks to save valuable time and pitfalls to avoid. Regardless of the size of your project, this book is for you.

This book was written for several audiences:

- **Software developers who want to learn to develop professional Android applications.** The bulk of this book is targeted at software developers with Java experience who do not necessarily have mobile development experience. More seasoned developers of mobile applications can learn how to take advantage of Android and how it differs from the other technologies of the mobile development market today.
- **Quality assurance personnel tasked with testing Android applications.** Whether they are black-box or white-box testing, quality assurance engineers can find this book invaluable. We devote several chapters to mobile QA concerns, including topics such as developing solid test plans and defect-tracking systems for mobile applications, how to manage handsets, and how to test applications thoroughly using all the Android tools available.
- **Project managers planning and managing Android development teams.** Managers can use this book to help plan, hire, and execute Android projects from start to finish. We cover project risk management and how to keep Android projects running smoothly.

- **Other audiences.** This book is useful not only to the software developer, but also to the corporation looking at potential vertical market applications, the entrepreneur thinking about a cool phone application, and the hobbyist looking for some fun with his or her new phone. Businesses seeking to evaluate Android for their specific needs (including feasibility analysis) can also find the information provided valuable. Anyone with an Android handset and a good idea for a mobile application can put the information provided in this book to use for fun and profit.

## Key Questions Answered in This Volume

This volume of the book answers the following questions:

1. What is Android? How do the SDK versions differ?
2. How is Android different from other mobile technologies, and how can developers take advantage of these differences?
3. How do developers use the Eclipse Development Environment for Java to develop and debug Android applications on the emulator and handsets?
4. How are Android applications structured?
5. How do developers design robust user interfaces for mobile—specifically, for Android?
6. What capabilities does the Android SDK have and how can developers use them?
7. How does the mobile development process differ from traditional desktop development?
8. What development strategies work best for Android development?
9. What do managers, developers, and testers need to look for when planning, developing, and testing a mobile development application?
10. How do mobile teams design bulletproof Android applications for publication?
11. How do mobile teams package Android applications for deployment?
12. How do mobile teams make money from Android applications?
13. And, finally, what is new in this edition of the book?

## How These Books Are Structured

We wrote the first edition of this book before the Android SDK was released. Now, three years and 14 Android SDK releases later, there is so much to talk about that we've had to divide the content of *Android Wireless Application Development* into two separate volumes for this, the third edition.

*Android Wireless Application Development Volume I: Android Essentials* focuses on Android essentials, including setting up your development environment, understanding the application lifecycle, user interface design, developing for different types of devices, and the mobile software process from design and development to testing and publication of commercial-grade applications.

*Android Wireless Application Development Volume II: Advanced Topics* focuses on advanced Android topics, including leveraging various Android APIs for threading, networking, location-based services, hardware sensors, animation, graphics, and more. Coverage of advanced Android application components, such as services, application databases, content providers, and intents, is also included. Developers learn to design advanced user interface components and integrate their applications deeply into the platform. Finally, developers learn how to extend their applications beyond traditional boundaries using optional features of the Android platform, including the Android Native Development Kit (NDK), Cloud-To-Device Messaging service (C2DM), Android Market In-Application Billing APIs, Google Analytics APIs, and more.

*Android Wireless Application Development Volume I: Android Essentials* is divided into six parts. The first four parts are primarily of interest to developers; Part V provides lots of helpful information for project managers and quality assurance personnel as well as developers. Part VI includes several helpful appendixes to help you get up and running with the most important Android tools.

Here is an overview of the various parts in this book:

- **Part I: An Overview of the Android Platform**

Part I provides an introduction to Android, explaining how it differs from other mobile platforms. You become familiar with the Android SDK and tools, install the development tools, and write and run your first Android application—on the emulator and on a handset.

- **Part II: Android Application Basics**

Part II introduces the design principles necessary to write Android applications. You learn how Android applications are structured and how to include resources, such as strings, graphics, and user interface components, in your projects.

- **Part III: Android User Interface Design Essentials**

Part III dives deeper into how user interfaces are designed in Android. You learn about the core user interface element in Android: the `View`. You also learn about the most common user interface controls and layouts provided in the Android SDK.

- **Part IV: Android Application Design Essentials**

Part IV covers the features used by most Android applications, including storing persistent application data using preferences and working with files, directories, and content providers. You also learn how to design applications that will run smoothly on many different Android devices.

- **Part V: Publishing and Distributing Android Applications**

Part V covers the software development process for mobile, from start to finish, with tips and tricks for project management, software developers, and quality assurance personnel.

- **Part VI: Appendixes**

Part VI includes two helpful quick-start guides for the Android development tools—the emulator and DDMS—as well as an appendix of Eclipse tips and tricks.

## An Overview of Changes in This Edition

When we began writing the first edition of this book, there were no Android devices on the market. One Android device became available shortly after we started writing, and it was available only in the United States. Today there are hundreds of devices shipping all over the world—smartphones, tablets, e-book readers, and specialty devices such as the Google TV. The Android platform has gone through extensive changes since the first edition of this book was published. The Android SDK has many new features and the development tools have received many much-needed upgrades. Android, as a technology, is now on solid footing within the mobile marketplace.

In this new edition, we took the opportunity to do a serious overhaul of the book content. But don't worry, it's still the book readers loved the first (and second!) time; it's just bigger, better, and more comprehensive. In order to cover more of the exciting topics available to Android developers, we had to divide the book into two volumes. In addition to adding tons of new content, we've retested and upgraded all existing content (text and sample code) for use with the latest Android SDKs available while still remaining backward compatible. The Android development community is diverse, and we aim to support all developers, regardless of which devices they are developing for. This includes developers who need to target nearly all platforms, so coverage in some key areas of older SDKs continues to be included because it's often the most reasonable option for compatibility.

Here are some of the highlights of the additions and enhancements we've made to this edition:

- Coverage of the latest and greatest Android tools and utilities.
- Updates to all existing chapters, often with some entirely new sections.
- New chapters, which cover new SDK features or expand upon those covered in previous editions.
- Updated sample code and applications, conveniently organized by chapter.
- Topics such as Android manifest files, content providers, designing apps, and testing now have their own chapters.

- Coverage of hot topics such as application compatibility, designing for different devices, and working with relatively new user interface components such as fragments.
- Even more tips and tricks from the trenches to help you design, develop, and test applications for different device targets, including an all-new chapter on tackling compatibility issues.

As you can see, we cover many of the hottest and most exciting features that Android has to offer. We didn't take this review lightly; we touched every existing chapter, updated content, and added many new chapters as well. Finally, we included many additions, clarifications, and, yes, even a few fixes based on the feedback from our fantastic (and meticulous) readers. Thank you!

## Development Environment Used in This Book

The Android code in this book was written using the following development environments:

- Windows 7 and Mac OS X 10.7.x
- Eclipse Java IDE Version 3.7 (Indigo) and Version 3.6 (Helios)
- Eclipse JDT plug-in and Web Tools Platform (WTP)
- Java SE Development Kit (JDK) 6 Update 26
- Android SDK Version 2.3.4, API Level 10 (Gingerbread MR1); Android SDK Version 3.2, API Level 13 (Honeycomb MR2); Android SDK Version 4.0, API Level 14 (Ice Cream Sandwich)
  1. ADT Plug-in for Eclipse 15.0.0
  2. SDK Tools Revision 15

Android Devices: Samsung Nexus S, HTC Evo 4G, Motorola Droid 3, Samsung Galaxy Tab 10.1, Motorola Xoom, Motorola Atrix 4G, and Sony Ericsson Xperia Play

The Android platform continues to aggressively grow in market share against competing mobile platforms, such as Apple iOS and BlackBerry. New and exciting types of devices reach consumers' hands at a furious pace, with new editions of the Android platform appearing all the time. Developers can no longer ignore Android as a target platform if they want to reach the smartphone (or smart-device) users of today and tomorrow.

Android's latest major platform update, Android 4.0—frequently called by its code-name, Ice Cream Sandwich, or just ICS—merges the smartphone-centric Android 2.3.x (Gingerbread) and the tablet-centric Android 3.x (Honeycomb) platform editions into a single SDK for all smart-devices, be they phones, tablets, televisions, or toasters. This

book features the latest SDK and tools available, but it does not focus on them to the detriment of popular legacy versions of the platform. This book is meant to be an overall reference to help developers support all popular devices on the market today. As of the writing of this book, only a very small percentage (less than 5%) of users' devices are running Android 3.0 or 4.0. Of course, some devices will receive upgrades, and users will purchase new Ice Cream Sandwich devices as they become available, but for now, developers need to straddle this gap and support numerous versions of Android to reach the majority of users in the field.

So what does this mean for this book? It means we provide both legacy API support and discuss some of the newer APIs only available in later versions of the Android SDK. We discuss strategies for supporting all (or at least most) users in terms of compatibility. And we provide screenshots that highlight different versions of the Android SDK, because each major revision has brought with it a change in the look and feel of the overall platform. That said, we are assuming that you are downloading the latest Android tools, so we provide screenshots and steps that support the latest tools available at the time of writing, not legacy tools. Those are the boundaries we set when trying to determine what to include or leave out of this book.

## Supplementary Materials Available

The source code that accompanies this book is available for download on the publisher website: [www.informit.com/title/9780321813831](http://www.informit.com/title/9780321813831). The source code is also available for download from our book website: <http://androidbook.blogspot.com/p/book-code-downloads.html> (<http://goo.gl/kyAsN>). You'll also find a variety of Android topics discussed at our book website (<http://androidbook.blogspot.com>). For example, we present reader feedback, questions, and further information. You can find links to our various technical articles on our book website as well.

## Where to Find More Information

There is a vibrant, helpful Android developer community on the Web. Here are a number of useful websites for Android developers and followers of the wireless industry:

- **Android Developer website:** The Android SDK and developer reference site:  
<http://developer.android.com> or <http://d.android.com>
- **Stack Overflow:** The Android website with great technical information (complete with tags) and an official support forum for developers:  
<http://stackoverflow.com/questions/tagged/android>
- **Open Handset Alliance:** Android manufacturers, operators, and developers:  
<http://www.openhandsetalliance.com>

- **Android Market:** Buy and sell Android applications:  
<http://www.android.com/market>
- **Mobiletuts+:** Mobile development tutorials, including Android:  
<http://mobile.tutsplus.com/category/tutorials/android>
- **anddev.org:** An Android developer forum:  
<http://www.anddev.org>
- **Google Team Android Apps:** Open-source Android applications:  
<http://apps-for-android.googlecode.com>
- **Android Tools Project Site:** The tools team discusses updates and changes:  
<https://sites.google.com/a/android.com/tools/recent>
- **FierceDeveloper:** A weekly newsletter for wireless developers:  
<http://www.fiercedeveloper.com>
- **Wireless Developer Network:** Daily news on the wireless industry:  
<http://www.wirelessdevnet.com>
- **XDA-Developers Android Forum:** From general development to ROMs:  
<http://forum.xda-developers.com/forumdisplay.php?f=564>
- **Developer.com:** A developer-oriented site with mobile articles:  
<http://www.developer.com>

## Conventions Used in This Book

This book uses the following conventions:

- ➔ is used to signify to readers that the authors meant for the continued code to appear on the same line. No indenting should be done on the continued line.
- Code or programming terms are set in monospace text.
- Java import statements, exception handling, and error checking are often removed from printed code examples for clarity and to keep the book a reasonable length.

This book also presents information in the following sidebars:

**Tip**

Tips provide useful information or hints related to the current text.

**Note**

Notes provide additional information that might be interesting or relevant.

**Warning**

Warnings provide hints or tips about pitfalls that may be encountered and how to avoid them.

## Contacting the Authors

We welcome your comments, questions, and feedback. We invite you to visit our blog at:

- <http://androidbook.blogspot.com>

Or email us at:

- [androidwirelessdev+awad3ev1@gmail.com](mailto:androidwirelessdev+awad3ev1@gmail.com)

Circle us on Google+:

- Lauren Darcey: <http://goo.gl/P3RGo>
- Shane Conder: <http://goo.gl/BpVjh>

# An Overview of the Android Platform

- 1** Introducing Android
- 2** Setting Up Your Android Development Environment
- 3** Writing Your First Android Application
- 4** Mastering the Android Development Tools

*This page intentionally left blank*

# Introducing Android

The mobile development community is at a tipping point. Mobile users demand more choices, more opportunities to customize their devices, and more functionality. Mobile operators want to provide value-added content to their subscribers in a manageable and lucrative way. Mobile developers want the freedom to develop the powerful mobile applications users demand with minimal roadblocks to success. Finally, handset manufacturers want a stable, secure, and affordable platform to power their devices.

Android has emerged as a game-changing platform for the mobile development community. An innovative and open platform, Android is well positioned to address the growing needs of the mobile marketplace as it continues to expand beyond early adopters and purchasers of high-end smart-devices.

This chapter explains what Android is, how and why it was developed, and where the platform fits into the established mobile marketplace. The first half of this chapter focuses on the history and the second half narrows in on how the Android platform operates.

## A Brief History of Mobile Software Development

To understand what makes Android so compelling, we must examine how mobile development has evolved and how Android differs from competing platforms.

### Way Back When

Remember way back when a phone was just a phone? When we relied on fixed land-lines? When we ran for the phone instead of pulling it out of our pocket? When we lost our friends at a crowded ballgame and waited around for hours hoping to reunite? When we forgot the grocery list (see Figure 1.1) and had to find a payphone or drive back home again?

Those days are long gone. Today, commonplace problems such as these are easily solved with a one-button speed dial or a simple text message such as “WRU?” or “20?” or “Milk and?”

Our mobile phones keep us safe and connected. Now we roam around freely, relying on our phones not only to keep us in touch with friends, family, and coworkers, but also to tell us where to go, what to do, and how to do it. Even the most domestic events seem to involve my mobile phone these days.



Figure 1.1 Mobile phones have become a crucial shopping accessory.

Consider the following true story, which has been slightly enhanced for effect:

*Once upon a time, on a warm summer evening, I was happily minding my own business cooking dinner in my new house in rural New Hampshire when a bat swooped over my head, scaring me to death.*

*The first thing I did—while ducking—was to pull out my cell phone and send a text message to my husband, who was across the country at the time. I typed, “There’s a bat in the house!”*

*My husband did not immediately respond (a divorce-worthy incident, I thought at the time), so I called my dad and asked him for suggestions on how to get rid of the bat.*

*He just laughed.*

*Annoyed, I snapped a picture of the bat with my phone and sent it to my husband and my blog, simultaneously guilt-tripping him and informing the world of my treacherous domestic wildlife encounter.*

*Finally, I googled “get rid of a bat” and then I followed the helpful do-it-yourself instructions provided on the Web for people in my situation. I also learned that late August is when*

*baby bats often leave the roost for the first time and learn to fly. Newly aware that I had a baby bat on my hands, I calmly got a broom and managed to herd the bat out of the house.*

*Problem solved—and I did it all with the help of my trusty cell phone, the old LG VX9800.*

My point here? Today's mobile devices aren't called smartphones for no reason. They can solve just about *anything*—and we rely on them for *everything* these days.

You notice that I used half a dozen different mobile applications over the course of this story. Each application was developed by a different company and had a different user interface. Some were well designed; others not so much. I paid for some of the applications, and others came on my phone.

As a user, I found the experience functional, but not terribly inspiring. As a mobile developer, I wished for an opportunity to create a more seamless and powerful application that could handle all I'd done and more. I wanted to build a better bat trap, if you will.

Before Android, mobile developers faced many roadblocks when it came to writing applications. Building the better application, the unique application, the competing application, the hybrid application, and incorporating many common tasks, such as messaging and calling, in a familiar way were often unrealistic goals.

To understand why, let's take a brief look at the history of mobile software development.

## **“The Brick”**

The Motorola DynaTAC 8000X was the first commercially available portable cell phone. First marketed in 1983, it was  $13 \times 1.75 \times 3.5$  inches in dimension, weighed about 2.5 pounds, and allowed you to talk for a little more than half an hour. It retailed for \$3,995, plus hefty monthly service fees and per-minute charges.

We called it “The Brick,” and the nickname stuck for many of those early mobile phones we alternately loved and hated. About the size of a brick, with battery power just long enough for half a conversation, these early mobile handsets were mostly seen in the hands of traveling business execs, security personnel, and the wealthy. First-generation mobile phones were just too expensive. The service charges alone would bankrupt the average person, especially when roaming.

Early mobile phones were not particularly full featured. (Although, even the Motorola DynaTAC, shown in Figure 1.2, had many of the buttons we've come to know well, such as the SEND, END, and CLR buttons.) These early phones did little more than make and receive calls and, if you were lucky, there was a simple contacts application that wasn't impossible to use.

The first-generation mobile phones were designed and developed by the handset manufacturers. Competition was fierce and trade secrets were closely guarded. Manufacturers didn't want to expose the internal workings of their handsets, so they usually developed the phone software in-house. As a developer, if you weren't part of this inner circle, you had no opportunity to write applications for the phones.

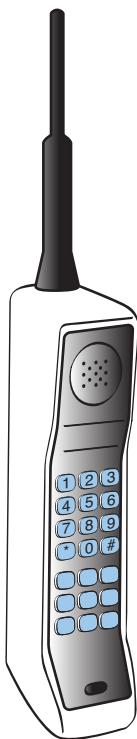


Figure 1.2 The first commercially available mobile phone:  
the Motorola DynaTAC.

It was during this period that we saw the first “time-waster” games begin to appear. Nokia was famous for putting the 1970s video game *Snake* on some of its earliest monochrome phones. Other manufacturers followed suit, adding games such as *Pong*, *Tetris*, and *Tic-Tac-Toe*.

These early devices were flawed, but they did something important—they changed the way people thought about communication. As mobile device prices dropped, batteries improved, and reception areas grew, more and more people began carrying these handy devices. Soon mobile devices were more than just a novelty.

Customers began pushing for more features and more games. But there was a problem. The handset manufacturers didn’t have the motivation or the resources to build every application users wanted. They needed some way to provide a portal for entertainment and information services without allowing direct access to the handset.

What better way to provide these services than the Internet?

## Wireless Application Protocol (WAP)

As it turned out, allowing direct phone access to the Internet didn't scale well for mobile.

By this time, professional websites were full color and chock full of text, images, and other sorts of media. These sites relied on JavaScript, Flash, and other technologies to enhance the user experience, and they were often designed with a target resolution of  $800 \times 600$  pixels and higher.

When the first clamshell phone, the Motorola StarTAC, was released in 1996, it merely had an LCD ten-digit segmented display. (Later models would add a dot-matrix-type display.) Meanwhile, Nokia released one of the first slider phones, the 8110—fondly referred to as “The Matrix Phone” because the phone was heavily used in films. The 8110 could display four lines of text with 13 characters per line. Figure 1.3 shows some of the common phone form factors.

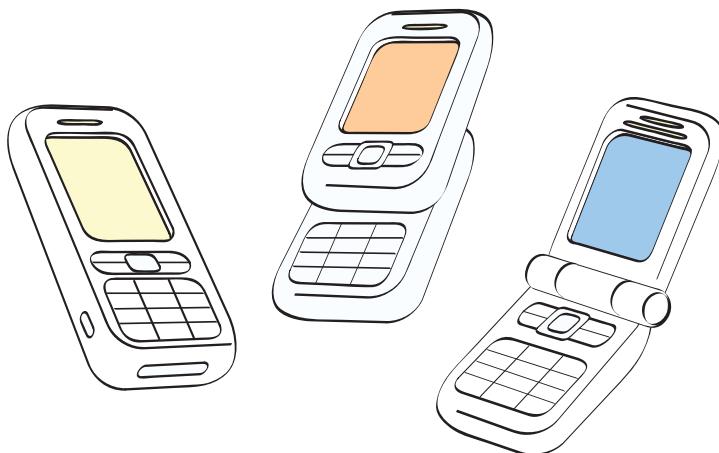


Figure 1.3 Various mobile phone form factors: the candy bar, the slider, and the clamshell.

With their postage stamp-sized low-resolution screens and limited storage and processing power, these phones couldn't handle the data-intensive operations required by traditional web browsers. The bandwidth requirements for data transmission were also costly to the user.

The Wireless Application Protocol (WAP) standard emerged to address these concerns. Simply put, WAP was a stripped-down version of HTTP, which is the backbone protocol of the World Wide Web. Unlike traditional web browsers, WAP browsers were designed to run within the memory and bandwidth constraints of the phone. Third-party WAP sites served up pages written in a markup language called Wireless Markup

Language (WML). These pages were then displayed on the phone's WAP browser. Users navigated as they would on the Web, but the pages were much simpler in design.

The WAP solution was great for handset manufacturers. The pressure was off—they could write one WAP browser to ship with the handset and rely on developers to come up with the content users wanted.

The WAP solution was great for mobile operators. They could provide a custom WAP portal, directing their subscribers to the content they wanted to provide, and rake in the data charges associated with browsing, which were often high.

For the first time, developers had a chance to develop content for phone users, and some did so, with limited success. Few gained any traction in the consumer markets because the content was of limited value and the end-user experience left much to be desired.

Most of the early WAP sites were extensions of popular branded websites, such as CNN.com and ESPN.com, which were looking for new ways to extend their readership. Suddenly phone users accessed the news, stock market quotes, and sports scores on their phones.

Commercializing WAP applications was difficult, and there was no built-in billing mechanism. Some of the most popular commercial WAP applications that emerged during this time were simple wallpaper and ringtone catalogues that enabled users to personalize their phones for the first time. For example, a user browsed a WAP site and requested a specific item. He filled out a simple order form with his phone number and his handset model. It was up to the content provider to deliver an image or audio file compatible with the given phone. Payment and verification were handled through various premium-priced delivery mechanisms such as Short Message Service (SMS), Enhanced Messaging Service (EMS), Multimedia Messaging Service (MMS), and WAP Push.

WAP browsers, especially in the early days, were slow and frustrating. Typing long URLs with the numeric keypad was onerous. WAP pages were often difficult to navigate. Most WAP sites were written one time for all phones and did not account for individual phone specifications. It didn't matter if the end user's phone had a big color screen or a postage stamp-sized monochrome screen; the developer couldn't tailor the user's experience. The result was a mediocre and not very compelling experience for everyone involved.

Content providers often didn't bother with a WAP site and instead just advertised SMS short codes on TV and in magazines. In this case, the user sent a premium SMS message with a request for a specific wallpaper or ringtone, and the content provider sent it back. Mobile operators generally liked these delivery mechanisms because they received a large portion of each messaging fee.

WAP fell short of commercial expectations. In some markets, such as Japan, it flourished, whereas in others, such as the United States, it failed to take off. Handset screens were too small for surfing. Reading a sentence fragment at a time, and then waiting seconds for the next segment to download, ruined the user experience, especially because

every second of downloading was often charged to the user. Critics began to call WAP “Wait and Pay.”

Finally, the mobile operators who provided the WAP portal (the default home page loaded when you started your WAP browser) often restricted which WAP sites were accessible. The portal enabled the operator to restrict the number of sites users could browse and to funnel subscribers to the operator’s preferred content providers and exclude competing sites. This kind of walled garden approach further discouraged third-party developers, who already faced difficulties in monetizing applications, from writing applications.

## Proprietary Mobile Platforms

It came as no surprise that users wanted more—they will always want more.

Writing robust applications with WAP, such as graphic-intensive video games, was nearly impossible. The 18-to-25-year-old sweet-spot demographic—the kids with the disposable income most likely to personalize their phones with wallpapers and ring-tones—looked at their portable gaming systems and asked for a device that was both a phone and a gaming device or a phone and a music player. They argued that if devices such as Nintendo’s Game Boy could provide hours of entertainment with only five buttons, why not just add phone capabilities? Others looked to their digital cameras, Palms, BlackBerries, iPods, and even their laptops and asked the same question. The market seemed to be teetering on the edge of device convergence.

Memory was getting cheaper, batteries were getting better, and PDAs and other embedded devices were beginning to run compact versions of common operating systems such as Linux and Windows. The traditional desktop application developer was suddenly a player in the embedded device market, especially with smartphone technologies such as Windows Mobile, which they found familiar.

Handset manufacturers realized that if they wanted to continue to sell traditional handsets, they needed to change their protectionist policies pertaining to handset design and expose their internal frameworks to some extent.

A variety of different proprietary platforms emerged—and developers are still actively creating applications for them. Some smartphone devices ran Palm OS (later known as Garnet OS) and RIM BlackBerry OS. Sun Microsystems took its popular Java platform and J2ME emerged (now known as Java Micro Edition [Java ME]). Chipset maker Qualcomm developed and licensed its Binary Runtime Environment for Wireless (BREW). Other platforms, such as Symbian OS, were developed by handset manufacturers such as Nokia, Sony Ericsson, Motorola, and Samsung. The Apple iPhone OS (OS X iPhone) joined the ranks in 2008. Figure 1.4 shows several different phones, all of which have different development platforms.

Many of these platforms have associated developer programs. These programs keep the developer communities small, vetted, and under contractual agreements on what they can and cannot do. These programs are often required and developers must pay for them.

Each platform has benefits and drawbacks. Of course, developers love to debate about which platform is “the best.” (Hint: It’s usually the platform we’re currently developing for.)

The truth is that no one platform has emerged victorious. Some platforms are best suited for commercializing games and making millions—if your company has brand backing. Other platforms are more open and suitable for the hobbyist or vertical market applications. No mobile platform is best suited for all possible applications. As a result, the mobile phone has become increasingly fragmented, with all platforms sharing part of the pie.



Figure 1.4 Phones from various mobile device platforms.

For manufacturers and mobile operators, handset product lines quickly became complicated. Platform market penetration varies greatly by region and user demographic. Instead of choosing just one platform, manufacturers and operators have been forced to sell phones for all the different platforms to compete in the market. We’ve even seen some handsets supporting multiple platforms. (For instance, Symbian phones often also support J2ME.)

The mobile developer community has become as fragmented as the market. It’s nearly impossible to keep track of all the changes in the market. Developer specialty niches have formed. The platform development requirements vary greatly. Mobile software developers work with distinctly different programming environments, different tools, and different programming languages. Porting among the platforms is often costly and not straightforward. Keeping track of handset configurations and testing requirements,

signing and certification programs, carrier relationships, and application marketplaces all have become complex spin-off businesses of their own.

It's a nightmare for the ACME Company that wants a mobile application. Should it develop a J2ME application? BREW? iPhone? Windows Mobile? Everyone has a different kind of phone. ACME is forced to choose one or, worse, all of the platforms. Some platforms allow for free applications, whereas others do not. Vertical market application opportunities are limited and expensive.

As a result, many wonderful applications have not reached their desired users, and many other great ideas have not been developed at all.

## The Open Handset Alliance

Enter search advertising giant Google. Now a household name, Google has shown an interest in spreading its vision, its brand, its search and ad revenue-based platform, and its suite of tools to the wireless marketplace. The company's business model has been amazingly successful on the Internet and, technically speaking, wireless isn't that different.

### Google Goes Wireless

The company's initial forays into mobile were beset with all the problems you would expect. The freedoms Internet users enjoyed were not shared by mobile phone subscribers. Internet users can choose from the wide variety of computer brands, operating systems, Internet service providers, and web browser applications.

Nearly all Google services are free and ad driven. Many applications in the Google Labs suite directly compete with the applications available on mobile phones. The applications range from simple calendars and calculators to navigation with Google Maps and the latest tailored news from News Alerts—not to mention corporate acquisitions such as Blogger and YouTube.

When this approach didn't yield the intended results, Google decided on a different approach—to revamp the entire system upon which wireless application development was based, hoping to provide a more open environment for users and developers: the Internet model. The Internet model allows users to choose between freeware, shareware, and paid software. This enables free-market competition among services.

### Forming the Open Handset Alliance

With its user-centric, democratic design philosophies, Google has led a movement to turn the existing closely guarded wireless market into one where phone users can move between carriers easily and have unfettered access to applications and services. With its vast resources, Google has taken a broad approach, examining the wireless infrastructure—from the FCC wireless spectrum policies to the handset manufacturers' requirements, application developer needs, and mobile operator desires.

Next, Google joined with other like-minded members in the wireless community and posed the following question: What would it take to build a better mobile phone?

The Open Handset Alliance (OHA) was formed in November 2007 to answer that very question. The OHA is a business alliance comprised of many of the largest and most successful mobile companies on the planet. Its members include chip makers, handset manufacturers, software developers, and service providers. The entire mobile supply chain is well represented.

Andy Rubin has been credited as the father of the Android platform. His company, Android Inc., was acquired by Google in 2005. Working together, OHA members, including Google, began developing a nonproprietary open-standard platform based on technology developed at Android Inc. that would aim to alleviate the aforementioned problems hindering the mobile community. The result is the Android project. To this day, most Android platform development is completed by Rubin's team at Google, where he acts as VP of Engineering and manages the Android platform roadmap.

Google's involvement in the Android project has been so extensive that the line between who takes responsibility for the Android platform (the OHA or Google) has blurred. Google provides the initial code for the Android open-source project and provides online Android documentation, tools, forums, and the Software Development Kit (SDK) for developers. Most major Android news originates from Google. The company has also hosted a number of events at conferences (Google IO, Mobile World Congress, CTIA Wireless) and the Android Developer Challenge (ADC), a series of contests to encourage developers to write killer Android applications—for millions of dollars in prizes to spur development on the platform. That's not to say they are the only ones involved, but they are the driving force behind the platform.

## Manufacturers: Designing Android Devices

More than half the members of the OHA are device manufacturers, such as Samsung, Motorola, Dell, Sony Ericsson, HTC, and LG, and semiconductor companies, such as Intel, Texas Instruments, ARM, NVIDIA, and Qualcomm.

The first shipping Android handset—the T-Mobile G1—was developed by handset manufacturer HTC with service provided by T-Mobile. It was released in October 2008. Many other Android handsets were slated for 2009 and early 2010. The platform gained momentum relatively quickly. By Q4, 2010, Android had come to dominate the smartphone market, gaining ground steadily against competitive platforms such as RIM BlackBerry, Apple iOS, and Windows Mobile.

Google normally announces Android platform statistics at its annual Google IO conference each May and at important events, such as financial earnings calls. As of July 2011, more than 310 Android devices were being shipped to 120 countries. More than 550,000 new Android devices were being activated every day, and more than 200 million devices have been activated (as of November 2011). The advantages of widespread manufacturer and carrier support appear to be really paying off at this point.

Manufacturers continue to create new generations of Android devices—from phones with HD displays, to watches for managing exercise programs, to dedicated e-book readers, to full-featured televisions, netbooks, and almost any other “smart” device you can imagine (see Figure 1.5).



Figure 1.5 A sampling of Android devices on the market today.

## Mobile Operators: Delivering the Android Experience

After you have the devices, you have to get them out to the users. Mobile operators from North, South, and Central America as well as Europe, Asia, India, Australia, Africa, and the Middle East have joined the OHA, ensuring a worldwide market for the Android movement. With almost half a billion subscribers alone, telephony giant China Mobile is a founding member of the alliance.

Much of Android's success is also due to the fact that many Android handsets don't come with the traditional "smartphone price tag"—quite a few are offered free with activation by carriers. Competitors such as the Apple iPhone have struggled to provide competitive offerings at the low-end of the market. For the first time, the average Jane or Joe can afford a feature-full smart-device. I've lost count of the number of times I've had a waitress, hotel night manager, or grocery store checkout person tell me that they just got an Android phone and it has changed their life. This phenomenon has only added to the Android's underdog status.

As of May 2011, more than 210 carriers and providers were selling Android devices worldwide. In the United States, the Android platform was given a healthy dose of help from carriers such as Verizon, Sprint, and T-Mobile. Verizon launched a \$100 million campaign for the first Droid handset, a series of devices that has now spanned several

generations of Android devices; a brand so strong many people think all Android phones are Droid phones. Sprint launched the Evo 4G to much fanfare and record one-day sales (<http://j.mp/uuWVIQ>). Carriers are now also providing subscribers with different types of devices, including competitive wireless plans for Android's newest tablets. For example, Samsung announced more than a million units sold of the first Galaxy Tab within its first quarter of availability and has since released several more devices in its Galaxy tablet lineup.

## Apps Drive Device Sales: Developing Android Applications

When users acquire Android devices, they need those killer apps, right?

Initially, Google led the pack in developing Android applications, many of which, such as the email client and web browser, are core features of the platform. They also developed the first successful distribution platform for third-party Android applications: the Android Market. The Android Market remains the primary method for which users download apps, but it is no longer the only distribution mechanism for Android apps.

As of July 2011, more than 250,000 applications were available in the Android Market, garnering more than 6 billion installations. This takes into account only applications published through this one marketplace—not the many other applications sold individually or on other markets. These numbers also do not take into account that newer Android devices support Flash applications, or all the web applications that target mobile devices running the Android platform. This opens up even more application choices for Android users and more opportunities for Android developers.

As of Google IO 2011, more than 450,000 registered Android developers were writing interesting and exciting applications. By the time you finish reading this book, you will be adding your expertise to this number.

## Taking Advantage of All Android Has to Offer

Android's open platform has been embraced by much of the mobile development community—extending far beyond the members of the OHA.

As Android devices and applications have become more readily available, many other mobile operators and device manufacturers have jumped at the chance to sell Android devices to their subscribers, especially given the cost benefits compared to proprietary platforms. The open standard of the Android platform has resulted in reduced operator costs in licensing and royalties, and we are now seeing a migration to more open devices from proprietary platforms such as RIM, Windows Mobile, and the Apple iPhone. The market has cracked wide open; new types of users are able to consider smartphones for the first time. Android is well suited to fill this demand.

## The Android Marketplace: Where We're at Now

The Android marketplace continues to grow at an aggressive rate on all fronts (devices, developers, and users). Lately, the focus has been on several topics:

- **Competitive hardware and software feature upgrades:** The Android SDK developers have focused on providing APIs for features that are available on competing platforms, to bring them into parity. For example, recent releases of the Android SDK have featured significant improvements from an enterprise perspective.
- **Expansion beyond smartphones:** Android was almost exclusively a smartphone platform prior to Android 3.x (Honeycomb). Android 4.0 (Ice Cream Sandwich) merges (dare we say “sandwiches”) the smartphone platform features and the tablet/other device features in one seamless SDK.
- **Free\*:** Android applications are free to develop. There are no licensing or royalty fees to develop on the platform. No required membership fees. No required testing fees. No required signing or certification fees. Android applications can be distributed and commercialized in a variety of ways. (The term “Free\*” implies there might actually be costs to development, but they are not mandated by the platform. Costs for designing, developing, testing, marketing, and maintaining are not included. If you provide all of these, you may not be laying out cash, but there is a cost associated with them.)
- **Improved user-facing features and marketing:** The Android development team has shifted focus from feature implementation to providing user-facing usability upgrades and “chrome.” Holographic user interfaces are the result. Google app developers have followed suit, updating the core apps to reflect better designs. The Android Market has received a number of sophisticated new features (including a web-based app store) and more interesting categories and organization.



### Note

Some may wonder about various legal battles surrounding Android that appear to involve almost every industry player in the mobile market. Although most of these issues do not affect developers directly, some have—in particular, those dealing with in-app purchases. This is typical of any popular platform. We can't provide any legal advice here. What we can recommend is keeping informed on the various legal battles and hope they turn out well, not just for Android, but for all platforms they impact.

## Android Platform Differences

The Android platform itself is hailed as “the first complete, open, and free mobile platform”:

- **Complete:** The designers took a comprehensive approach when they developed the Android platform. They began with a secure operating system and built a robust software framework on top that allows for rich application development opportunities.

- **Open:** The Android platform is provided through open-source licensing. Developers have unprecedented access to the device features when developing applications.
- **Free:** Android applications are free to develop. There are no licensing or royalty fees to develop on the platform. No required membership fees. No required testing fees. No required signing or certification fees. Android applications can be distributed and commercialized in a variety of ways.

## Android: A Next-Generation Platform

Although Android has many innovative features not available in existing mobile platforms, its designers also leveraged many tried-and-true approaches proven to work in the wireless world. It's true that many of these features appear in existing proprietary platforms, but Android combines them in a free and open fashion while simultaneously addressing many of the flaws on these competing platforms.

The Android mascot is a little green robot, shown in Figure 1.6. This little guy (girl?) is often used to depict Android-related materials.



Figure 1.6 Some Android SDKs and their codenames.

Android is the first in a new generation of mobile platforms, giving its platform developers a distinct edge on the competition. Android's designers examined the benefits and drawbacks of existing platforms and then incorporated their most successful features. At the same time, Android's designers avoided the mistakes others suffered in the past.

Since the Android 1.0 SDK was released, Android platform development has continued at a fast and furious pace. For quite some time, a new Android SDK came out every couple of months! In typical tech-sector jargon, each Android SDK has had a project name. In Android's case, the SDKs are named alphabetically after sweets (see Figure 1.6).

The latest version of Android is codenamed Ice Cream Sandwich.

## **Free and Open Source**

Android is an open-source platform. Neither developers nor device manufacturers pay royalties or license fees to develop for the platform.

The underlying operating system of Android is licensed under GNU General Public License Version 2 (GPLv2), a strong “copyleft” license where any third-party improvements must continue to fall under the open-source licensing agreement terms. The Android framework is distributed under the Apache Software License (ASL/Apache2), which allows for the distribution of both open- and closed-source derivations of the source code. Commercial developers (device manufacturers especially) can choose to enhance the platform without having to provide their improvements to the open-source community. Instead, developers can profit from enhancements such as device-specific improvements and redistribute their work under whatever licensing they want.

Android application developers have the ability to distribute their applications under whatever licensing scheme they prefer. Developers can write open-source freeware or traditional licensed applications for profit and everything in between.

## **Familiar and Inexpensive Development Tools**

Unlike some proprietary platforms that require developer registration fees, vetting, and expensive compilers, there are no upfront costs to developing Android applications.

### **Freely Available Software Development Kit**

The Android SDK and tools are freely available. Developers can download the Android SDK from the Android website after agreeing to the terms of the Android Software Development Kit License Agreement.

### **Familiar Language, Familiar Development Environments**

Developers have several choices when it comes to integrated development environments (IDEs). Many developers choose the popular and freely available Eclipse IDE to design and develop Android applications. Eclipse is the most popular IDE for Android development, and an Android plug-in is available for facilitating Android development. Android applications can be developed on the following operating systems:

- Windows XP (32-bit), Windows Vista (32-bit or 64-bit), and Windows 7 (32-bit or 64-bit)
- Mac OS X 10.5.8 or later (x86 only)
- Linux (tested on Ubuntu Linux 10.04 LTS, Lucid Lynx, 8.04 LTS or later is required)

## **Reasonable Learning Curve for Developers**

Android applications are written in a well-respected programming language: Java.

The Android application framework includes traditional programming constructs, such as threads and processes and specially designed data structures to encapsulate objects commonly used in mobile applications. Developers can rely on familiar class libraries such as `java.net` and `java.text`. Specialty libraries for tasks such as graphics and database management are implemented using well-defined open standards such as OpenGL Embedded Systems (OpenGL ES) and SQLite.

## **Enabling Development of Powerful Applications**

In the past, device manufacturers often established special relationships with trusted third-party software developers (OEM/ODM relationships). This elite group of software developers wrote native applications, such as messaging and web browsers, that shipped on the device as part of its core feature set. To design these applications, the manufacturer would grant the developer privileged inside access and knowledge of the internal software framework and firmware.

On the Android platform, there is no distinction between native and third-party applications, thus helping maintain healthy competition among application developers. All Android applications use the same APIs. Android applications have unprecedented access to the underlying hardware, allowing developers to write much more powerful applications. Applications can be extended or replaced altogether.

## **Rich, Secure Application Integration**

Recall from the bat story I previously shared that I accessed a variety of phone applications in the course of a few moments: text messaging, phone dialer, camera, email, picture messaging, and the browser. Each was a separate application running on the phone—some built in and some purchased. Each had its own unique user interface. None were truly integrated.

Not so with Android. One of the Android platform's most compelling and innovative features is well-designed application integration. Android provides all the tools necessary to build a better “bat trap,” if you will, by allowing developers to write applications that seamlessly leverage core functionality such as web browsing, mapping, contact management, and messaging. Applications can also become content providers and share their data among each other in a secure fashion.

Platforms such as Symbian have suffered from setbacks due to malware. Android's vigorous application security model helps protect the user and the system from malicious software.

## No Costly Obstacles to Publication

Android applications have none of the costly and time-intensive testing and certification programs required by other platforms such as BREW and Symbian. No costly developer programs are required to start developing for the Android platform. All you need is a computer, an Android device, a good idea, an understanding of Java, and perhaps (if we may be so bold) this book (Volume 2 wouldn't hurt, either).

## A “Free Market” for Applications

Android developers are free to choose any kind of revenue model they want. They can develop freeware, shareware, trial-ware or ad-driven applications, and paid applications. Android was designed to fundamentally change the rules about what kind of wireless applications could be developed. In the past, developers faced many restrictions that had little to do with the application functionality or features:

- Store limitations on the number of competing applications of a given type
- Store limitations on pricing, revenue models, and royalties
- Operator unwillingness to provide applications for smaller demographics

With Android, developers can write and successfully publish any kind of application they want. Developers can tailor applications to small demographics, instead of just large-scale money-making ones often insisted upon by mobile operators. Vertical market applications can be deployed to specific, targeted users.

Because developers have a variety of application distribution mechanisms to choose from, they can pick the methods that work for them instead of being forced to play by others' rules. Android developers can distribute their applications to users in a variety of ways:

- Google developed the Android Market, a generic Android application store with a revenue-sharing model. Android Market now has a web store for browsing and buying apps online (see Figure 1.7). Android Market also sells movies, music, and books, so your application will be in good company.
- Amazon Appstore for Android launched in 2011 with a lineup of exciting Android applications using its own billing and revenue-sharing models. A unique feature of Amazon Appstore for Android is that it allows users to demo certain applications in the web browser before purchasing them. Amazon uses an environment similar to the emulator that runs right in the browser.
- Numerous other third-party application stores are available. Some are for niche markets; others cater to many different mobile platforms.
- Developers can come up with their own delivery and payment mechanisms, such as distributing from a website or within an enterprise.

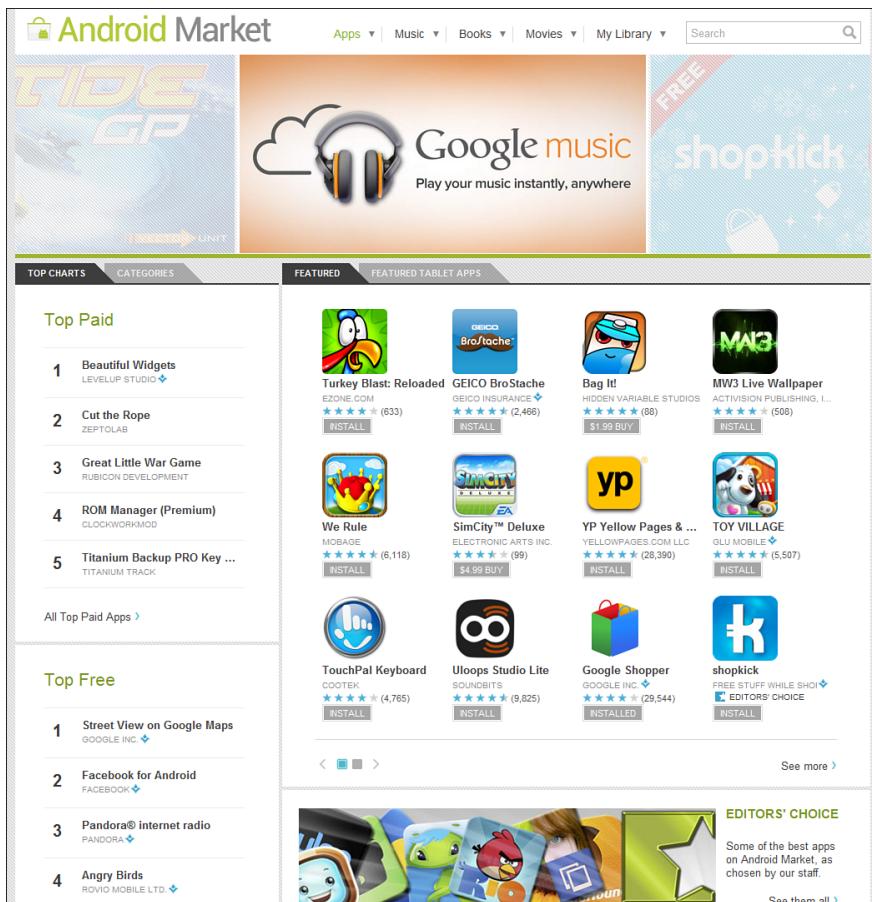


Figure 1.7 The Android Market online, showing apps.

Mobile operators and carriers are still free to develop their own application stores and enforce their own rules, but this will no longer be the only opportunity developers have to distribute their applications. Be sure to read any application store agreements carefully before distributing your applications on them.

## A Growing Platform

Early Android developers have had to deal with the typical roadblocks associated with a new platform: frequently revised SDKs, lack of good documentation, and market uncertainties. Some older Android devices are not capable of running the latest versions of the platform. This means that Android developers often need to target several different SDK versions to reach all users. Luckily, the continuously evolving Android development tools

have made this easier than ever, and now that Android is a well-established platform, many of these issues have been ironed out. The Android forum community is lively and friendly and very supportive when it comes to helping one another over these bumps in the road.

Each new version of the Android SDK has provided a number of substantial improvements to the platform. In recent revisions, the Android platform has received some much-needed UI “polish,” both in terms of visual appeal and performance. New types of devices such as tablets and Internet TVs are now fully supported by the platform.

Although most of these upgrades and improvements were welcome and necessary, new SDK versions often cause some upheaval within the Android developer community. A number of published applications have required retesting and resubmission to the Android Marketplace to conform to new SDK requirements, which are quickly rolled out to all Android devices in the field as a firmware upgrade, rendering older applications obsolete and sometimes unusable.

Although these growing pains are expected, and most developers have rolled with them, it’s important to remember that Android was a latecomer to the mobile marketplace compared to the RIM and iOS platforms. The Apple App Store still boasts more applications, but users are demanding these same applications on their Android devices, so it is only a matter of time before these markets equalize. Fewer developers can afford to deploy exclusively to one platform or the other—they must support all the popular ones. Developers diving into Android development now can still benefit from the first-to-market competitive advantages we’ve seen on other platforms.

## The Android Platform

Android is an operating system and a software platform upon which applications are developed. A core set of applications for everyday tasks, such as web browsing and email, are included on Android devices.

As a product of the OHA’s vision for a robust and open-source development environment for wireless, Android is an emerging mobile development platform. The platform was designed for the sole purpose of encouraging a free and open market that users might want to have and software developers might want to develop for.

## Android’s Underlying Architecture

The Android platform is designed to be more fault-tolerant than many of its predecessors. The device runs a Linux operating system upon which Android applications are executed in a secure fashion. Each Android application runs in its own virtual machine (see Figure 1.8). Android applications are managed code; therefore, they are much less likely to cause the device to crash, leading to fewer instances of device corruption (also called “brickling” the device, or rendering it useless).

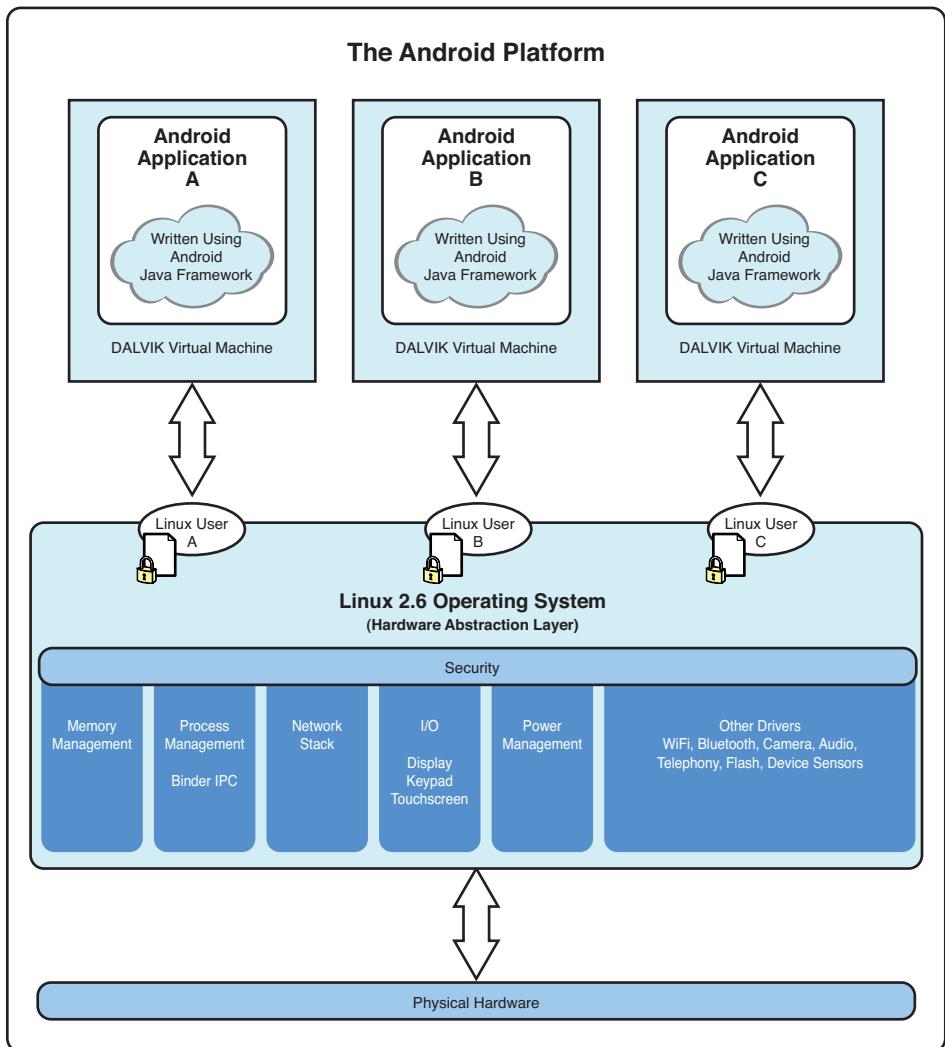


Figure 1.8 Diagram of the Android platform architecture.

### The Linux Operating System

The Linux 2.6 kernel handles core system services and acts as a hardware abstraction layer (HAL) between the physical hardware of the device and the Android software stack.

Some of the core functions the kernel handles include

- Enforcement of application permissions and security
- Low-level memory management
- Process management and threading
- The network stack
- Display, keypad input, camera, Wi-Fi, Flash memory, audio, and binder (IPC) driver access

## **Android Application Runtime Environment**

Each Android application runs in a separate process, with its own instance of the Dalvik virtual machine (VM). Based on the Java VM, the Dalvik design has been optimized for mobile devices. The Dalvik VM has a small memory footprint and optimized application loading, and multiple instances of the Dalvik VM can run concurrently on the device.

## **Security and Permissions**

The integrity of the Android platform is maintained through a variety of security measures. These measures help ensure that the user's data is secure and that the device is not subjected to malware.

### **Applications as Operating System Users**

When an application is installed, the operating system creates a new user profile associated with the application. Each application runs as a different user, with its own private files on the file system, a user ID, and a secure operating environment.

The application executes in its own process with its own instance of the Dalvik VM and under its own user ID on the operating system.

### **Explicitly Defined Application Permissions**

To access shared resources on the system, Android applications register for the specific privileges they require. Some of these privileges enable the application to use device functionality to make calls, access the network, and control the camera and other hardware sensors. Applications also require permission to access shared data containing private and personal information, such as user preferences, user's location, and contact information.

Applications might also enforce their own permissions by declaring them for other applications to use. The application can declare any number of different permission types, such as read-only or read-write permissions, for finer control over the application.

### **Limited Ad-Hoc Permissions**

Applications that act as content providers might want to provide some on-the-fly permissions to other applications for specific information they want to share openly. This is done using ad-hoc granting and revoking of access to specific resources using Uniform Resource Identifiers (URIs).

URIs index-specific data assets on the system, such as images and text. Here is an example of a URI that provides the phone numbers of all contacts:

```
content://contacts/phones
```

To understand how this permission process works, let's look at an example.

Let's say we have an application that keeps track of the user's public and private birthday wish lists. If this application wanted to share its data with other applications, the application could grant URI permissions for the public wish list, allowing another application permission to access this list without explicitly having to ask the user for the list.

### **Application Signing for Trust Relationships**

All Android applications packages are signed with a certificate, so users know that the application is authentic. The private key for the certificate is held by the developer. This helps establish a trust relationship between the developer and the user. It also enables the developer to control which applications can grant access to one another on the system. No certificate authority is necessary; self-signed certificates are acceptable.

### **Marketplace Developer Registration**

To publish applications on the popular Android Market, developers must create a developer account. The Android Market is managed closely and no malware is tolerated.

## **Developing Android Applications**

The Android SDK provides an extensive set of application programming interfaces (APIs) that is both modern and robust. Android device core system services are exposed and accessible to all applications. When granted the appropriate permissions, Android applications can share data among one another and access shared resources on the system securely.

### **Android Programming Language Choices**

Android applications are written in Java (see Figure 1.9). For now, the Java language is the developer's only choice for accessing the entire Android SDK.



#### **Tip**

There has been some speculation that other programming languages, such as C++, might be added in future versions of Android. If your application must rely on native code in another language such as C or C++, you might want to consider integrating it using the Android Native Development Kit (NDK). We talk more about how to use the NDK in the second volume of this book.

You can also develop mobile web applications that will run on Android devices. These applications can be accessed through the Android Browser application, or an embedded `WebView` control within a native Android application (still written in Java). We discuss the `WebView` control in this book and in *Android Wireless Application Development Volume*

*II: Advanced Topics*, but leave web app development to those who specialize in web technologies such as HTML5. This book focuses on native application development. You can find out more about developing web applications for Android devices at the Android developer website: <http://goo.gl/jYeOe>.

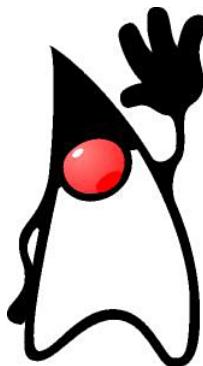


Figure 1.9 Duke, the Java mascot.

Got a Flash app you want to deploy to the Android platform? Check out Adobe's AIR support for the Android platform. Users install the Adobe AIR application from the Android Market and then load your compatible applications using it. For more information, see the Adobe website: <http://goo.gl/b31wT>. As with web apps, developing Flash applications are outside the scope of this book.

### No Distinctions Made Between Native and Third-Party Applications

Unlike other mobile development platforms, there is no distinction between native applications and developer-created applications on the Android platform. Provided they are granted the appropriate permissions, all applications have the same access to core libraries and the underlying hardware interfaces.

Android devices ship with a set of native applications such as a web browser and contact manager. Third-party applications might integrate with these core applications, extend them to provide a rich user experience, or replace them entirely with alternative applications. The idea is that any of these applications is built using the exact same APIs available to third-party developers, thus ensuring a level playing field, or as close to one as we can get.

Note that although this has been Google's line since the beginning, there are some cases where Google has used undocumented APIs. Because Android is open, there are no private APIs. They have never blocked access to such APIs, but have warned developers that using them may present incompatibilities in future SDK versions. See the blog post at <http://goo.gl/ad6PK> for some recent examples of APIs that have become publicly documented that were previously undocumented.

## Commonly Used Packages

With Android, mobile developers no longer have to reinvent the wheel. Instead, developers use familiar class libraries exposed through Android's Java packages to perform common tasks involving graphics, database access, network access, secure communications, and utilities. The Android packages include support for the following:

- A wide variety of user interface controls (Buttons, Spinners, Text input)
- A wide variety of user interface layouts (Tables, Tabs, Lists, Galleries)
- Secure networking and web browsing features (SSL, WebKit)
- XML support (DOM, SAX, XMLPullParser)
- Structured storage and relational databases (App Preferences, SQLite)
- Powerful 2D and 3D graphics (including SGL, OpenGL ES, and RenderScript)
- Multimedia frameworks for playing and recording standalone or network streaming (MediaPlayer, JetPlayer, SoundPool)
- Extensive support for many audio and visual media formats (MPEG4, MP3, Still Images)
- Access to optional hardware such as location-based services (LBS), USB, Wi-Fi, Bluetooth, and hardware sensors

## Android Application Framework

The Android application framework provides everything necessary to implement your average application. The Android application lifecycle involves the following key components:

- Activities are functions the application performs.
- Groups of views define the application's layout.
- Intents inform the system about an application's plans.
- Services allow for background processing without user interaction.
- Notifications alert the user when something interesting happens.
- Content providers facilitate data transmission among different applications.

## Android Platform Services

Android applications can interact with the operating system and underlying hardware using a collection of managers. Each manager is responsible for keeping the state of some underlying system service. For example:

- The `LocationManager` facilitates interaction with the location-based services available on the device.
- The `ViewManager` and `WindowManager` manage display and user interface fundamentals related to the device.

- The `AccessibilityManager` manages accessibility events, facilitating device support for users with physical impairments.
- The `ClipboardManager` provides access to the global clipboard for the device, for cutting and pasting content.
- The `AudioManager` provides access to audio and ringer controls.

## Summary

Mobile software development has evolved over time. Android has emerged as a new mobile development platform, building on past successes and avoiding past failures of other platforms. Android was designed to empower the developer to write innovative applications. The platform is open source, with no up-front fees, and developers enjoy many benefits over other competing platforms. Now it's time to dive deeper and start writing Android code, so you can evaluate what Android can do for you.

## References and More Information

Android Development:

<http://developer.android.com>

Open Handset Alliance:

<http://www.openhandsetalliance.com>

Official Android Developers Blog:

<http://android-developers.blogspot.com>

This Book's blog:

<http://androidbook.blogspot.com>

*This page intentionally left blank*

# Setting Up Your Android Development Environment

Android developers write and test applications on their computers and then deploy those applications onto the actual device hardware for further testing.

In this chapter, you become familiar with all the tools you need to master in order to develop Android applications. You also explore the Android Software Development Kit (SDK) installation and all it has to offer.



## Note

The Android development tools are updated frequently. We have made every attempt to provide the latest steps for the latest tools. However, these steps and the user interfaces described in this chapter may change at any time. Please review the Android development website (<http://d.android.com/sdk>) and our book website (<http://androidbook.blogspot.com>) for the latest information.

## Configuring Your Development Environment

To write Android applications, you must configure your programming environment for Java development. The software is available online for download at no cost. Android applications can be developed on Windows, Macintosh, or Linux systems.

To develop Android applications, you need to have the following software installed on your computer:

- The **Java Development Kit (JDK)**, Version 5 or 6, available for download at <http://www.oracle.com/technetwork/java/javase/downloads/index.htm> (or <http://java.sun.com/javase/downloads/index.jsp>, if you're nostalgic).
- A compatible **Java IDE**, such as **Eclipse**, along with its JDT plug-in. Eclipse is available for download at <http://www.eclipse.org/downloads/>. You will need Eclipse 3.5 or later and either Eclipse IDE for Java Developers, Eclipse Classic, or Eclipse IDE for Java EE Developers.

- The latest **Android SDK**, tools and documentation, available for download at <http://d.android.com/sdk/index.html> (starter packages for Windows, Linux, and Mac, including an installer for Windows).
- The **Android Development Tools plug-in for Eclipse (ADT)**, available for download through the Eclipse software update mechanism. For instructions on how to install this plug-in, see <http://d.android.com/sdk/eclipse-adt.html>. Although this tool is optional for development, we highly recommend it and will use its features frequently throughout this book.

A complete list of Android development system requirements is available at <http://d.android.com/sdk/requirements.html>.



### Tip

Most developers use the popular Eclipse integrated development environment (IDE) for Android development. The Android development team has integrated the Android development tools directly into the Eclipse IDE. However, developers are not constrained to using Eclipse; they can also use other IDEs. For information on using other development environments, begin by reading <http://d.android.com/guide/developing/projects/projects-cmdline.html>, which talks about using the command line tools, which may be useful for using other environments. In addition, read <http://d.android.com/guide/developing/debugging/debugging-projects-cmdline.html> for information on debugging from other IDEs and [http://d.android.com/guide/developing/testing/testing\\_otheride.html](http://d.android.com/guide/developing/testing/testing_otheride.html) for testing from other IDEs.

The basic installation process follows these steps:

1. Download and install the appropriate JDK.
2. Download and install the appropriate version of Eclipse (or other compatible IDE). Check for any updates and patches and install these as well.
3. Download and install the Android SDK Starter package.
4. (Eclipse users only.) Download and install the Android ADT Plug-in for Eclipse. Configure the Android settings in the Eclipse preferences. Make sure you specify the directory where you installed the Android SDK and save your settings.
5. Use the Android SDK Manager to download and install specific Android platform versions and other components you might use, including the documentation, sample applications, USB drivers, and additional tools. The Android SDK Manager is available as a standalone tool in the Android SDK Starter package, as well as from within Eclipse once you have installed the plug-in in the previous step. In terms of which components you'll want to choose, we recommend a full installation (choose everything).
6. Configure your computer for device debugging by installing the appropriate USB drivers, if necessary.

7. Configure your Android device(s) for device debugging.
8. Launch Eclipse (or your preferred IDE) and start developing Android applications.

For the purposes of this book, we do not give you detailed step-by-step instructions for installing each and every component listed in the preceding steps for three main reasons. First, this is an intermediate/advanced book and we expect you have some familiarity with installing Java development tools and SDKs. Second, the Android Developer website provides fairly extensive information for installing development tools and configuring them on a variety of different operating systems. Installation instructions are available at <http://d.android.com/sdk/installing.html>. Third, the exact steps required to install the Android SDK tend to change subtly with each and every major release (and some minor releases) so you're always better off checking the Android Developer website for the latest information. Finally, keep in mind that the Android SDK and tools are updated frequently and may not exactly match the development environment used in this book, as defined in the Introduction. That said, we will help you work through some of the later steps in the process described in this section, starting after you've installed and configured the JDK, Eclipse, the Android SDK, and the ADT plug-in in Step 6. We'll poke around in the Android SDK and look at some of the core tools you'll need to use to develop applications. Then, in the next chapter, you'll test your development environment and write your first Android application.

## **Configuring Your Operating System for Device Debugging**

To install and debug Android applications on Android devices, you need to configure your operating system to access the phone via the USB cable (see Figure 2.1). On some operating systems, such as Mac OS, this may just work. However, for Windows installations, you need to install the appropriate USB driver. You can download the Windows USB driver from the following website: <http://d.android.com/sdk/win-usb.html>. Under Linux, there are some additional steps to perform. See <http://d.android.com/guide/developing/device.html> for more information.

## **Configuring Your Android Hardware for Debugging**

Android devices have debugging disabled by default. Your Android device must be enabled for debugging via a USB connection to allow the tools to install and launch the applications you deploy.

You need to enable your device to install Android applications other than those from the Android Market. This setting is reached by selecting Home, Menu, Settings, Applications. Here you should check (enable) the option called Unknown Sources, as shown in Figure 2.2. If you do not enable this option, you cannot install developer-created applications, the sample applications, or applications published on alternative markets without the developer tools. Loading applications from servers or even email is a great way to test deployments.

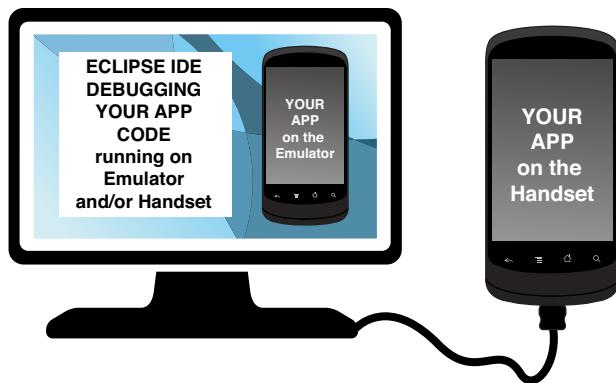


Figure 2.1 Android application debugging using the emulator and an Android handset.

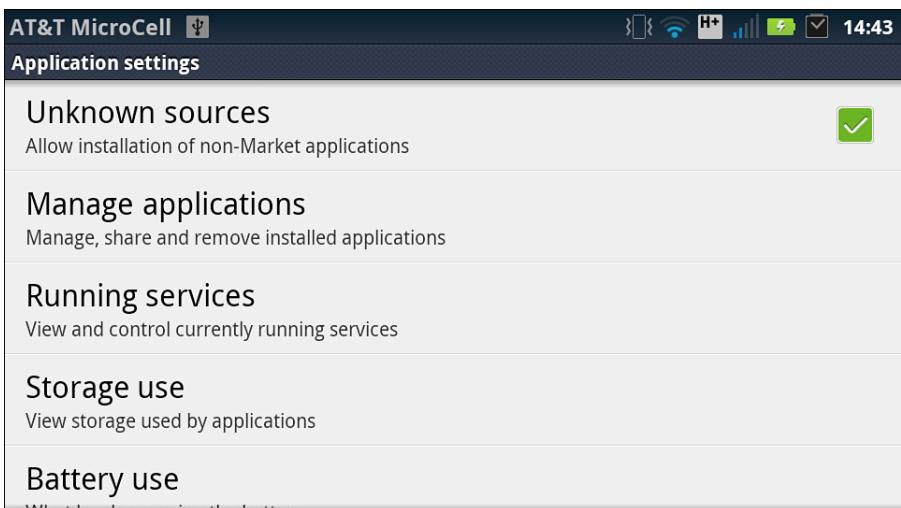


Figure 2.2 Enabling unknown sources on the device.

Several other important development settings are available on the Android device by selecting Home, Menu, Settings, Applications, Development (see Figure 2.3). Here you should enable the following options:

- **USB Debugging:** This setting enables you to debug your applications via the USB connection.
- **Stay Awake:** This convenient setting keeps the phone from sleeping in the middle of your development work, as long as the device is plugged in.

- **Allow Mock Locations:** This setting enables you to send mock location information to the phone for development purposes and is very convenient for applications using location-based services (LBS).

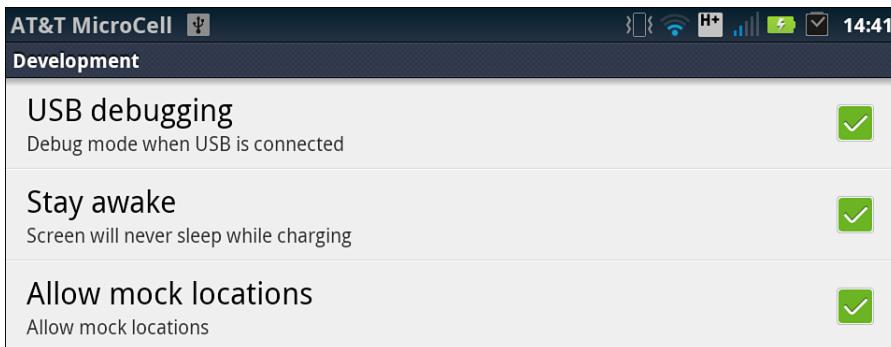


Figure 2.3 Enabling Android developer settings on the device.

## Upgrading the Android SDK

The Android SDK is upgraded from time to time. You can easily upgrade the Android SDK and tools from within Eclipse using the Android SDK Manager, which is installed as part of the ADT plug-in for Eclipse.

Changes to the Android SDK might include addition, update, and removal of features; package name changes; and updated tools. With each new version of the SDK, Google provides the following useful documents:

- **An Overview of Changes:** A brief description of the major changes to the SDK
- **An API Diff Report:** A complete list of specific changes to the SDK
- **Release Notes:** A list of known issues with the SDK

These documents are available with every new release of the Android SDK. For instance, Android 2.2 information is available at <http://d.android.com/sdk/android-2.2.html> and Android 4.0 information is available at <http://d.android.com/sdk/android-4.0.html>.

You can find out more about adding and updating SDK components at <http://d.android.com/sdk/adding-components.html>.

## Problems with the Android Software Development Kit

Because the Android SDK is constantly under active development, you might come across problems with the SDK. If you think you've found a problem, you can find a list of open issues and their status at the Android project's Issue Tracker website. You can also submit new issues for review.

The Issue Tracker website for the Android open-source project is <http://code.google.com/p/android/issues/list>. For more information about logging your own bugs or defects to be considered by the Android platform development team, check out the following website: <http://source.android.com/source/report-bugs.html>.



### Tip

Frustrated with how long it takes for your bug to get fixed? It can be helpful to understand how the Android bug-resolution process works. For more information on this process, see the following website: <http://source.android.com/source/life-of-a-bug.html>.

## Exploring the Android SDK

The Android SDK comes with five major components: the Android SDK License Agreement, the Android Documentation, Application Framework, Tools, and Sample Applications.

### Understanding the Android SDK License Agreement

Before you can download the Android SDK, you must review and agree to the Android SDK License Agreement. This agreement is a contract between you (the developer) and Google (copyright holder of the Android SDK).

Even if someone at your company has agreed to the Licensing Agreement on your behalf, it is important for you, the developer, to be aware of a few important points:

- **Rights granted:** Google (as the copyright holder of Android) grants you a limited, worldwide, royalty-free, non-assignable, and non-exclusive license to use the SDK solely to develop applications for the Android platform. Google (and third-party contributors) are granting you license, but they still hold all copyrights and intellectual property rights to the material. Using the Android SDK does not grant you permission to use any Google brands, logos, or trade names. You will not remove any of the copyright notices therein. Third-party applications that your applications interact with (other Android apps) are subject to separate terms and fall outside this agreement.
- **SDK usage:** You may only develop Android applications. You may not make derivative works from the SDK or distribute the SDK on any device or distribute part of the SDK with other software.
- **SDK changes and backward compatibility:** Google may change the Android SDK at any time, without notice, without regard to backward compatibility. Although Android API changes were a major issue with prerelease versions of the

SDK, recent releases have been reasonably stable. That said, each SDK update does tend to affect a small number of existing applications in the field, thus necessitating updates.

- **Android application developer rights:** You retain all rights to any Android software you develop with the SDK, including intellectual property rights. You also retain all responsibility for your own work.
- **Android application privacy requirements:** You agree that your applications will protect the privacy and legal rights of its users. If your application uses or accesses personal and private information about the user (usernames, passwords, and so on), then your application will provide an adequate privacy notice and keep that data stored securely. Note that privacy laws and regulations may vary by user location; you as a developer are solely responsible for managing this data appropriately.
- **Android application malware requirements:** You are responsible for all applications you develop. You agree not to write disruptive applications or malware. You are solely responsible for all data transmitted through your application.
- **Additional terms for specific Google APIs:** Use of the Android Maps API is subject to further Terms of Service (specifically use of the following packages: `com.google.android.maps` and `com.android.location.Geocoder`). You must agree to these additional terms before using those specific APIs and always include the Google Maps copyright notice provided. Use of Google Data APIs (Google apps such as Gmail, Blogger, Google Calendar, Google Finance Portfolio Data, Picasa, YouTube, and so on) is limited to access that the user has explicitly granted permission to your application by accepted permissions provided by the developer during installation time.
- **Develop at your own risk:** Any harm that comes about from developing with the Android SDK is your own fault and not Google's.

## Reading the Android SDK Documentation

The Android documentation is provided in HTML format locally and online at <http://d.android.com>. A local copy of the Android documentation is provided in the `docs` subfolder of the Android installation directory (as shown in Figure 2.4).

## Exploring the Core Android Application Framework

The Android application framework is provided in the `android.jar` file. The Android SDK is made up of several important packages, which are shown in Table 2.1.

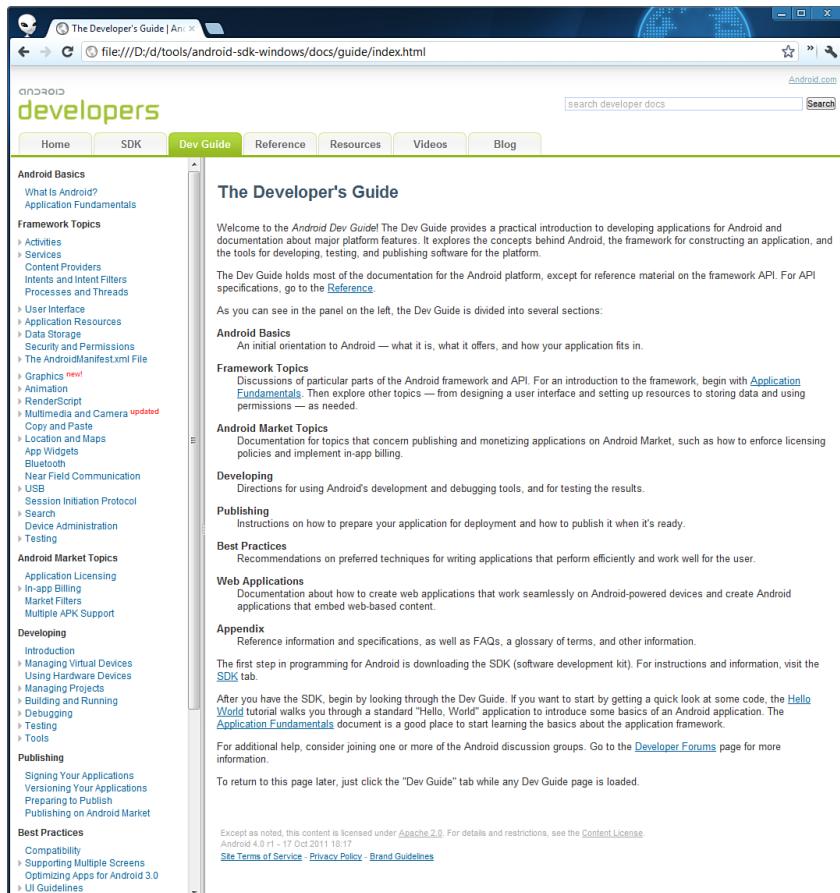


Figure 2.4 The Android SDK documentation.

Table 2.1 Important Packages in the Android SDK

Top Level Package Name	Description
android.*	Android application fundamentals
dalvik.*	Dalvik virtual machine support classes
java.*	Core classes and familiar generic utilities for networking, security, math, and so on
javax.*	Encryption support
junit.*	Unit testing support

Top Level Package Name	Description
org.apache.http.*	Hypertext Transfer Protocol (HTTP) protocol support
org.json	JavaScript Object Notation (JSON) support
org.w3c.dom	W3C Java bindings for the Document Object Model Core (XML and HTML)
org.xml.sax.*	Simple API for XML (SAX) support for XML
org.xmlpull.*	High-performance XML pull parsing

Several optional third-party APIs are available outside the core Android SDK. These packages must be installed separately from their respective websites. Some packages are from Google, whereas others are from device manufacturers and other providers. Some of the most popular third-party APIs are described in Table 2.2.

Table 2.2 Popular Third-Party Android APIs

Optional Android SDKs	Description
Google APIs add-on	Facilitates development using Google Maps and other Google APIs and services. For example, if you want to include the <code>MapView</code> control in your application, you need to install and use this feature. This add-on requires agreement to additional Terms of Service and registration for an API Key. For more info, see <a href="http://code.google.com/android/add-ons/google-apis/">http://code.google.com/android/add-ons/google-apis/</a> .
Android Support Package	Adds several components available in recent SDKs to legacy versions of the SDKs. For example, the Loader APIs and Fragment APIs introduced in API Level 11 can be used, in compatibility form, as far back as API Level 4 using this add-on.
Packages: Various	
Android Cloud to Device Messaging (C2DM)	Provides access to a service for developers to push data from the network to their applications installed on devices. This SDK requires agreement to additional Terms of Service and registration for an account. For more info, see <a href="http://code.google.com/android/c2dm/">http://code.google.com/android/c2dm/</a> .
Package: com.google.android.c2dm	
Google Analytics SDK for Android	Allows developers to collect and analyze information about how their Android applications are used with the popular Google Analytics service. This SDK requires agreement to additional Terms of Service and registration for an account. For more information, see <a href="http://code.google.com/mobile/analytics/docs/android/">http://code.google.com/mobile/analytics/docs/android/</a> .
Package: com.google.android.apps.analytics	

Table 2.2 **Continued**

Optional Android SDKs	Description
Google Market Billing  Package: <code>com.android.vending.billing</code>	Allows developers of applications targeting the Android Market to enable in-application purchases. This SDK requires agreement to additional Terms of Service and must be tied to your Android Market publisher account. For more info, see <a href="http://d.android.com/guide/market/billing/">http://d.android.com/guide/market/billing/</a> .
Google Market Licensing  Package: <code>com.android.vending</code>	Allows developers of applications targeting the Android Market to enable in-application license verification. This SDK requires agreement to additional Terms of Service and must be tied to your Android Market publisher account. For more information, see <a href="http://d.android.com/guide/publishing/licensing.html">http://d.android.com/guide/publishing/licensing.html</a> .
Numerous device and manufacturer-specific add-ons and SDKs	You'll find a number of third-party add-ons and SDKs available within the Android SDK and AVD Manager's Available Packages. Still others can be found at third-party websites. If you are targeting features available from a specific device or manufacturer, or services from a known service provider, check to see if they have add-ons available for the Android platform. For example, Motorola and AuthenTec provide an add-on SDK for accessing the fingerprint reader on their Motorola ATRIX 4G smartphone on their developer website.

## Exploring the Core Android Tools

The Android SDK provides many tools to design, develop, debug, and deploy your Android applications. For now, we want you to focus on familiarizing yourself with the core tools you need to know about to get up and running Android applications. We'll discuss many of Android tools in greater detail in Chapter 4, "Mastering the Android Development Tools."

### Eclipse and the ADT Plug-in

You'll spend most of your development time in your IDE. This book assumes you are using Eclipse with the ADT plug-in, because this is the most popular development environment configuration.

The ADT plug-in for Eclipse incorporates many of the most important Android SDK tools seamlessly into your development environment and provides various wizards for creating, debugging, and deploying Android applications. The ADT plug-in adds a

number of useful functions to the default Eclipse IDE. Several new buttons are available on the toolbar, including buttons to perform the following actions:

- Launch the Android SDK Manager
- Launch the Android Virtual Device Manager
- Create a new project using the Android Project Wizard
- Create a new test project using the Android Project Wizard
- Create a new Android XML resource file

Eclipse organizes its workspace into perspectives (each a set of specific panes) for different tasks such as coding and debugging. You can switch between perspectives by choosing the appropriate tab in the top-right corner of the Eclipse environment. The Java perspective arranges the appropriate panes for coding and navigating around the project. The Debug perspective enables you to set breakpoints, view LogCat information, and debug. The ADT plug-in adds several special Eclipse perspectives for designing and debugging Android applications. The Hierarchy View perspective integrates the Hierarchy Viewer tool into Eclipse so that you can design, inspect, and debug user interface controls within your applications. The DDMS perspective integrates the Dalvik Debug Monitor Server (DDMS) tool into Eclipse so that you can attach to emulator and device instances and debug your applications. Figure 2.5 shows the Eclipse toolbar with the added Android toolbar features (left/middle) and the new Android perspectives (right).

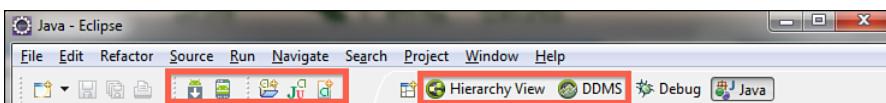


Figure 2.5 Android features added to the Eclipse toolbar and perspectives.

You can switch perspectives within Eclipse by choosing Window, Open Perspective or by clicking the appropriate perspective tab from the top-right corner of the Eclipse toolbar.

### Android SDK and AVD Managers

The first Android toolbar icon with the little green Android and the down arrow will launch the Android SDK Manager (see Figure 2.6, top). The second Android toolbar icon, looking like a tiny phone, will launch the Android Virtual Device Manager (see Figure 2.6, bottom).

These tools perform two major functions: management of Android SDK components installed on the development machine and management of the developer's Android Virtual Device (AVD) configurations.

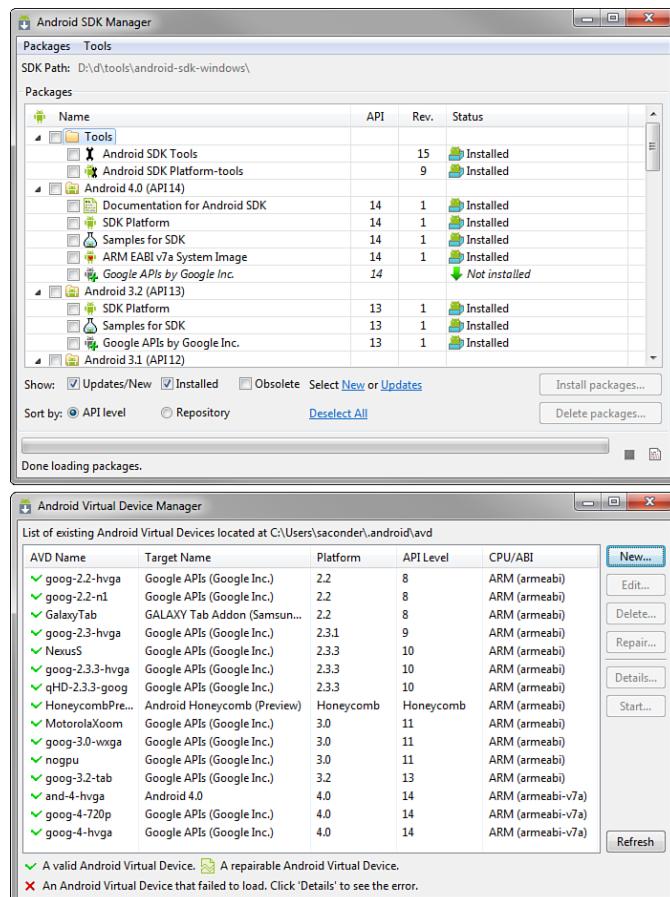


Figure 2.6 The Android SDK Manager and  
Android Virtual Device Manager.

Much like desktop computers, different Android devices run different versions of the Android operating system. Developers need to be able to target different Android SDK versions with their applications. Some applications target a specific Android SDK, whereas others try to provide simultaneous support for as many versions as possible.

The Android SDK Manager facilitates Android development across multiple platform versions simultaneously. When a new Android SDK is released, you can use this tool to download and update your tools while still maintaining backward compatibility and use older versions of the Android SDK.

The Android Virtual Device Manager organizes and provides tools to create and edit AVDs. To manage applications in the Android emulator, you must configure different AVD profiles. Each AVD profile describes what type of device you want the emulator to

simulate, including which Android platform to support as well what the device specifications should be. You can specify different screen sizes and orientations, and you can specify whether the emulator has an SD card and, if so, what capacity, among many other device configuration settings.

## Android Emulator

The Android emulator is one of the most important tools provided with the Android SDK. You will use this tool frequently when designing and developing Android applications. The emulator runs on your computer and behaves much as a mobile device would. You can load Android applications into the emulator, test, and debug them.

The emulator is a generic device and is not tied to any one specific phone configuration. You describe the hardware and software configuration details that the emulator is to simulate by providing an AVD configuration. Figure 2.7 shows what the emulator might look like with a typical Android 2.3.3 smartphone-style AVD configuration.

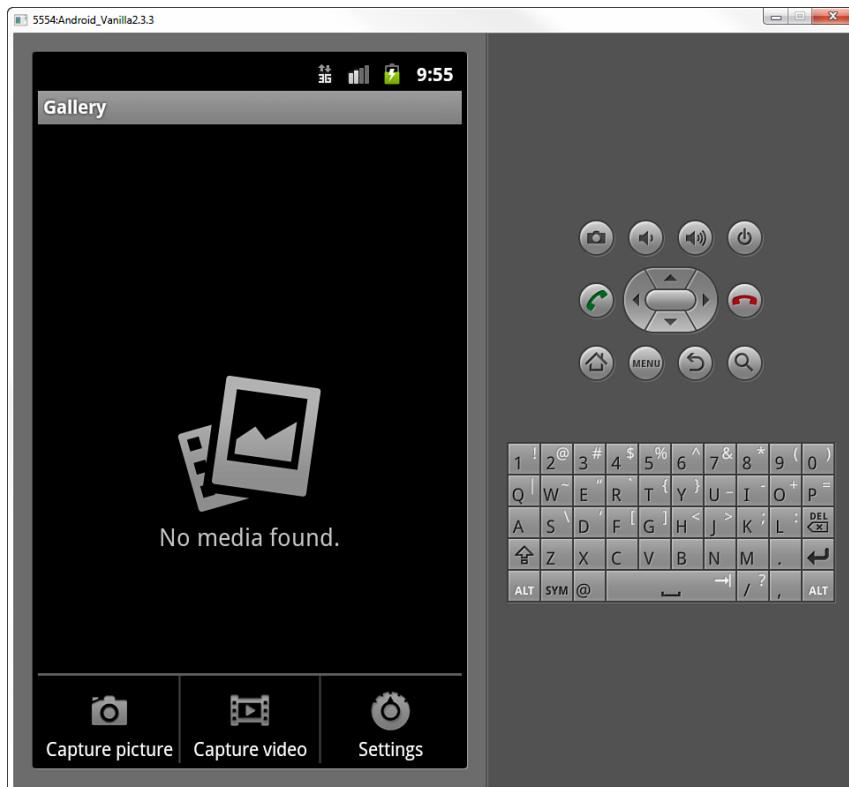


Figure 2.7 The Android emulator (smartphone-style, Gingerbread AVD configuration).

Figure 2.8 shows what the emulator might look like with a typical Android 3.2 tablet-style AVD configuration. Both Figures 2.7 and 2.8 show how the popular Gallery application behaves differently on different devices.

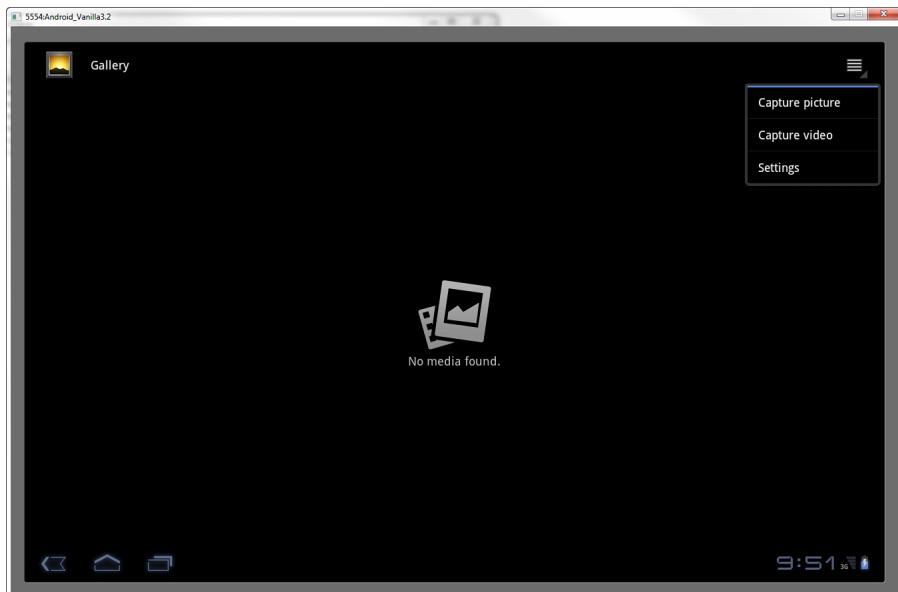


Figure 2.8 The Android emulator (tablet-style, Honeycomb AVD configuration).



### Tip

You should be aware that the Android emulator is a substitute for a real Android device, but it's an imperfect one. The emulator is a valuable tool for testing but cannot fully replace testing on actual target devices.

## Exploring the Android Sample Applications

The Android SDK provides many samples and demo applications to help you learn the ropes of Android development. Many of these demo applications are provided as part of the Android SDK and are located in the `/samples` subdirectory of the Android SDK.

More than two dozen sample applications are available to demonstrate different aspects of the Android SDK. Some focus on generic application development tasks such as managing application lifecycle or user interface design.



### Tip

On some Android SDK installations, the sample applications must be downloaded separately by updating your SDK installation using the Android SDK Manager. All sample applications can also be found on the Android Developer website.

Some of the most straightforward demo applications to take a look at are

- **ApiDemos:** A menu-driven utility that demonstrates a wide variety of Android APIs, from user interface widgets to application lifecycle components such as services, alarms, and notifications
- **Snake:** A simple game that demonstrates bitmap drawing and key events
- **NotePad:** A simple list application that demonstrates database access and Live Folder functionality
- **LunarLander:** A simple game that demonstrates drawing and animation
- **HoneycombGallery:** A simple application that demonstrates the various features available in Android 3.0+
- **Spinner and SpinnerTest:** A simple application that demonstrates some application lifecycle basics, as well as how to create and manage application test cases using the unit test framework
- **TicTacToeMain and TicTacToeLib:** A simple application that demonstrates how to create and manage shared code libraries for use with your applications

There are numerous other sample applications that cover specific topics, but they demonstrate Android features that are discussed later in this book. You can find more information about the sample applications on the Android Developer website under the Resources tab. Some other sample apps are only found online. Some of the most useful online sample applications to take a look at are listed here:

- **Support4Demos:** A sample application that demonstrates the key features in the Android API Level 4+ compatibility library, including fragments and loaders. You can find out more about this sample application at the Android Developer website: <http://d.android.com/resources/samples/Support4Demos/index.html>.
- **Support13Demos:** A sample application that demonstrates the key features in the Android API Level 13+ compatibility library, including enhanced fragment support. You can find out more about this sample application at the Android Developer website: <http://d.android.com/resources/samples/Support13Demos/index.html>.



### Tip

To add a sample project within Eclipse, select File, New Android Project and then choose the build target. This causes the “Create project from existing sample” radio button to become available. The sample applications available depend on which specific build target is chosen. Proceed as you normally would, by creating a Debug Configuration, compiling and running the application in the emulator or on a device. You will see these steps performed in detail in the next chapter when you test your development environment and write your first application.

## Summary

In this chapter, you installed, configured, and began to explore the tools you need to start developing Android applications, including the appropriate JDK, the Eclipse development environment, and the Android SDK. You explored many of the tools provided along with the Android SDK and understand their basic purposes. Finally, you perused the sample applications provided along with the Android SDK. You should now have a reasonable development environment configured to write Android applications. In the next chapter, you’ll be able to take advantage of all this setup and write an Android application.

## References and More Information

Google’s Android Developer’s Guide:

<http://d.android.com/guide/index.html>

Android SDK download site:

<http://d.android.com/sdk>

Android SDK License Agreement:

<http://d.android.com/sdk/terms.html>

The Java Platform, Standard Edition:

<http://www.oracle.com/technetwork/java/javase>

The Eclipse Project:

<http://www.eclipse.org>

# 3

## Writing Your First Android Application

You should now have a workable Android development environment set up on your computer. Hopefully, you also have an Android device as well. Now it's time for you to start writing some Android code. In this chapter, you learn how to add and create Android projects in Eclipse and verify that your Android development environment is set up correctly. You also write and debug your first Android application in the software emulator and on an Android device.



### Note

The Android development tools are updated frequently. We have made every attempt to provide the latest steps for the latest tools. However, these steps and the user interfaces described in this chapter may change at any time. Please review the Android development website (<http://d.android.com/sdk>) and our book website (<http://androidbook.blogspot.com>) for the latest information.

## Testing Your Development Environment

The best way to make sure you configured your development environment correctly is to take an existing Android application and run it. You can do this easily by using one of the sample applications provided as part of the Android SDK in the `samples` subdirectory found where your Android SDK is installed.

Within the Android SDK sample applications, you will find a classic game called *Snake* (<http://goo.gl/wRojX>). To build and run the *Snake* application, you must create a new Android project in your Eclipse workspace based on the existing Android sample project, create an appropriate Android Virtual Device (AVD) profile, and configure a launch configuration for that project. After you have everything set up correctly, you can build the application and run it on the Android emulator and on an Android device. By testing your development environment with a sample application, you can rule out project configuration and coding issues and focus on determining whether the tools are set

up properly for Android development. After this fact has been established, you can move on to writing and compiling your own applications.

## Adding the Snake Project to Your Eclipse Workspace

The first thing you need to do is add the Snake project to your Eclipse workspace. To do this, follow these steps:

1. Choose File, New, Other....
2. Choose Android, Android Sample Project (see Figure 3.1). Click Next.

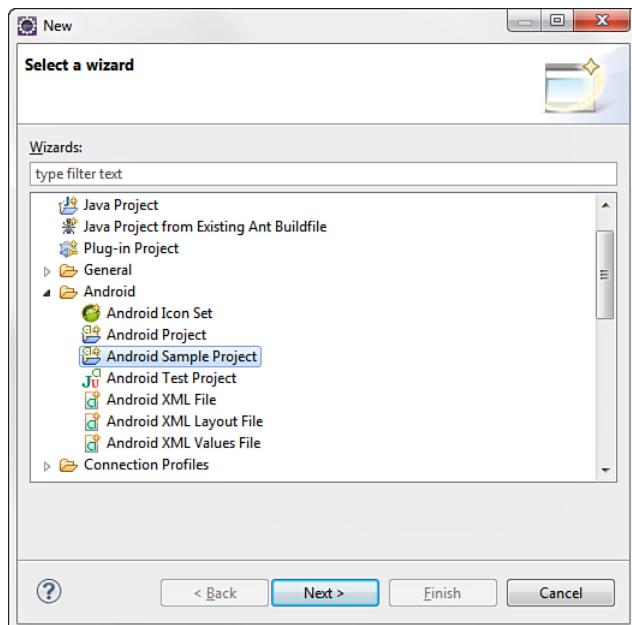


Figure 3.1 Creating a new Android project.

3. Choose your build target (see Figure 3.2). In this case, we've picked Android 4.0, API Level 14, from the Android Open Source Project. Click Next.
4. Next, select which sample you want to create (see Figure 3.3). Choose Snake.
5. Click Finish. You now see the Snake project files in your workspace (see Figure 3.4).

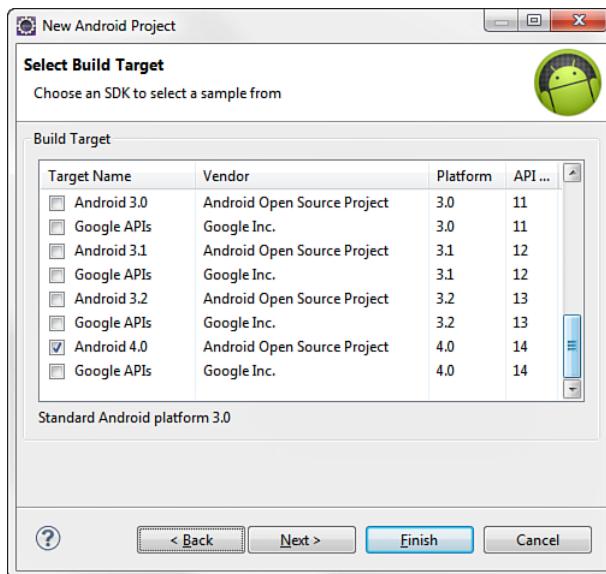


Figure 3.2 Choose an API level for the sample.

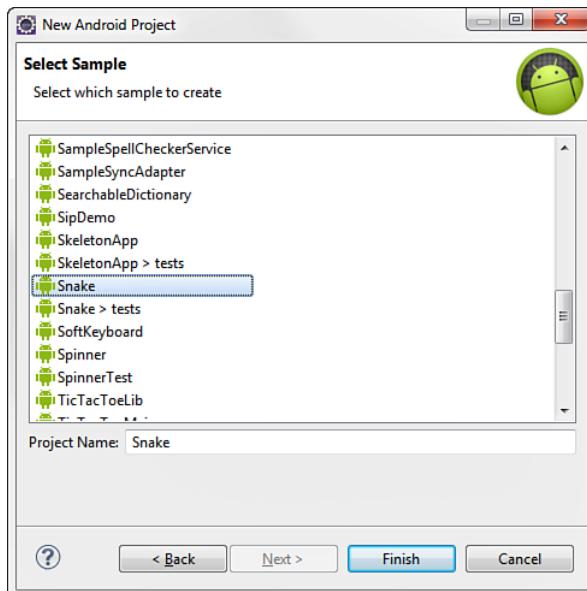


Figure 3.3 Picking the sample project.

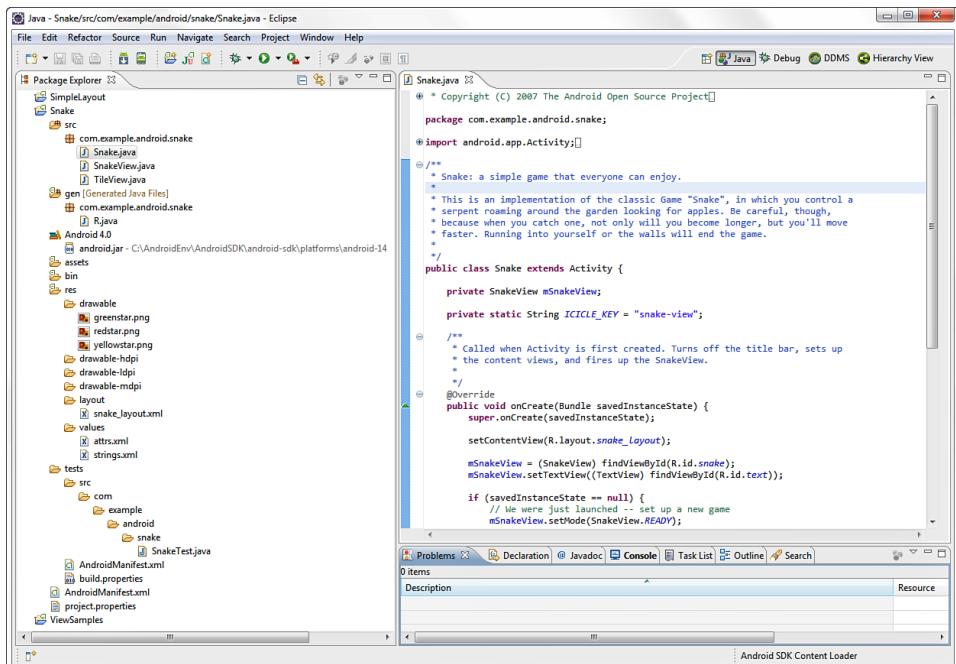


Figure 3.4 The Snake project files.



### Warning

Occasionally Eclipse shows the error “Project ‘Snake’ is missing required source folder: gen” when you’re adding an existing project to the workspace. If this happens, navigate to the project file called R.java under the /gen directory and delete it. The R.java file is automatically regenerated and the error should disappear. Performing a Clean operation followed by a Build operation does not always solve this problem.

## Creating an Android Virtual Device (AVD) for Your Snake Project

The next step is to create an AVD that describes what type of device you want to emulate when running the Snake application. This AVD profile describes what type of device you want the emulator to simulate, including which Android platform to support. You do not need to create new AVDs for each application, only for each device you want to emulate. You can specify different screen sizes and orientations, and you can specify whether the emulator has an SD card and, if it does, what capacity the card is.

For the purposes of this example, an AVD for the default installation of Android 2.3.3 suffices. Here are the steps to create a basic AVD:

1. Launch the Android Virtual Device Manager from within Eclipse by clicking the little Android device icon on the toolbar (). If you cannot find the icon, you can also launch the manager through the Window menu of Eclipse.
2. Click the New button.
3. Choose a name for your AVD. Because we are going to take all the defaults, give this AVD a name of `Android_Vanilla4.0`.
4. Choose a build target. We want a typical Android 4.0 device, so choose Google APIs (Google Inc.) – API Level 14 from the drop-down menu. This will include the Google Android applications, such as the Maps application, as part of the platform image.
5. Choose an SD card capacity, in either kibibytes or mibibytes. Not familiar with kibibytes? See this Wikipedia entry: <http://goo.gl/N3Rdd>. This SD card image will take up space on your hard drive, so choose something reasonable, such as 1024MiB.
6. Seriously consider enabling the Snapshot feature. This feature greatly improves emulator startup performance. See Appendix A, “The Android Emulator Quick-Start Guide,” for details.

### Warning

As of this writing, there's a known issue with the Android Tools R14 emulator that prevents the snapshot feature from working correctly. See <http://goo.gl/pnMt0> for more information on the current known issues.

7. Choose a skin. This option controls the different resolutions of the emulator. In this case, we use the recommended WVGA800 screen. (The emulator in Tools R14 has known performance issues with running the standard Android 4.0 screen resolution of WXGA720.) This skin most directly correlates to the popular 4.0 devices. Feel free to choose the most appropriate skin to match the Android device on which you plan to run the application.

Your project settings will look like Figure 3.5.

8. Click the Create AVD button and then wait for the operation to complete.
9. Click Finish. Because the AVD manager formats the memory allocated for SD card images, creating AVDs with SD cards could take a few moments.

For more information on creating different types of AVDs, check out Appendix A.

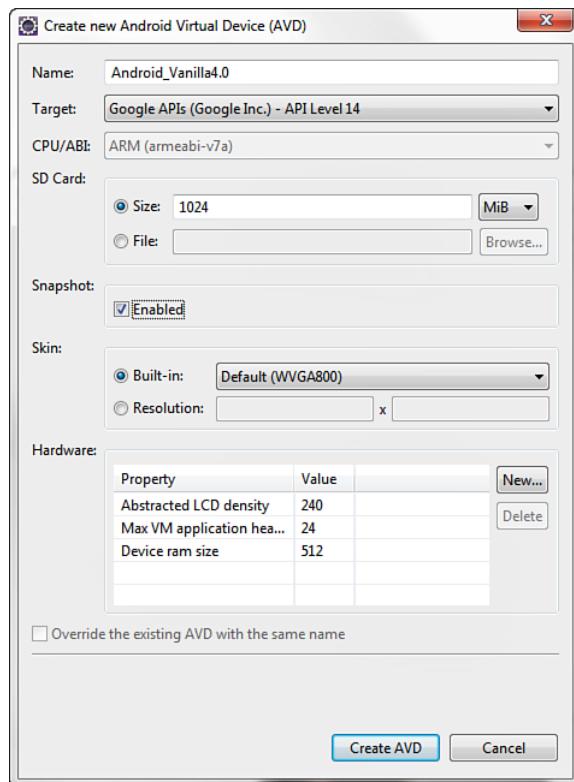


Figure 3.5 Creating a new AVD.

## Creating a Launch Configuration for Your Snake Project

Next, you must create a launch configuration in Eclipse to configure under what circumstances the Snake application builds and launches. The launch configuration is where you configure the emulator options to use and the entry point for your application.

You can create Run configurations and Debug configurations separately, each with different options. These configurations are created under the Run menu in Eclipse (Run, Run Configurations and Run, Debug Configurations). Follow these steps to create a basic Debug configuration for the Snake application:

1. Choose Run, Debug Configurations.
2. Double-click Android Application to create a new configuration.
3. Name your Debug configuration `SnakeDebugConfiguration`.
4. Choose the project by clicking the Browse button and choosing the Snake project.

5. Switch to the Target tab and, from the preferred AVD list, choose the `Android_Vanilla4.0` AVD created earlier, as shown in Figure 3.6.

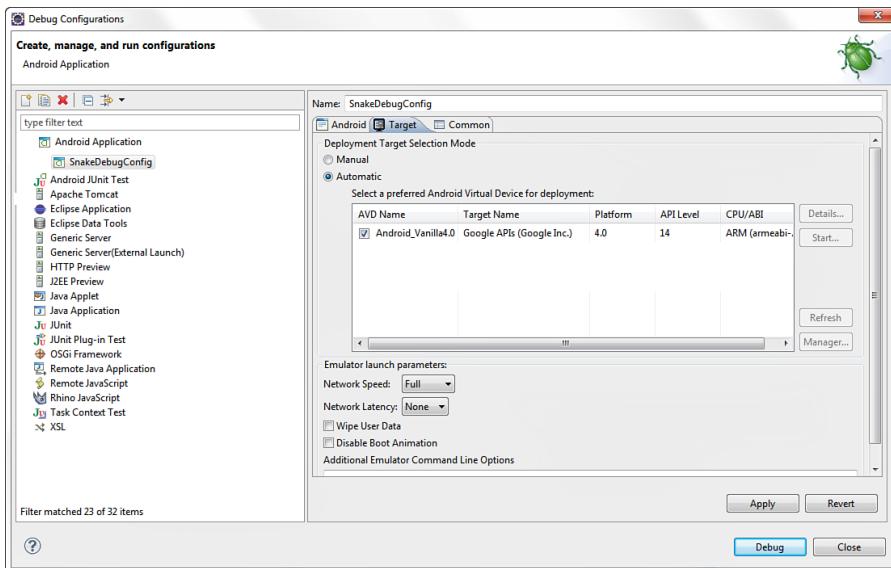


Figure 3.6 The Snake application Debug configuration in Eclipse.

You can set other emulator and launch options on the Target and Common tabs, but for now we are leaving the defaults as they are.

## Running the Snake Application in the Android Emulator

Now you can run the Snake application using the following steps:

1. Choose the Debug As icon drop-down menu on the toolbar ().
2. Pull the drop-down menu and choose the `SnakeDebugConfiguration` you created. If you do not see your new configuration listed, find it in the Debug Configurations listing and click the Debug button. Subsequent launches can be initiated from the little bug drop-down.
3. The Android emulator starts up; this might take a few moments to initialize. Then the application will be installed or reinstalled onto the emulator.



### Tip

It can take a long time for the emulator to start up, even on very fast computers. You might want to leave it around while you work and reattach to it as needed. The tools in Eclipse handle reinstalling the application and re-launching the application, so you can more easily keep the emulator loaded all the time. This is another reason to enable the Snapshot feature for each AVD. You can also use the Start button on the Android Virtual Device Manager to load up an emulator before you need it. Launching the AVD this way also gives you some additional options such as screen scaling (see Figure 3.7), which can be used to either fit the AVD on your screen if it's very high resolution or more closely emulate the size it might be on real hardware.

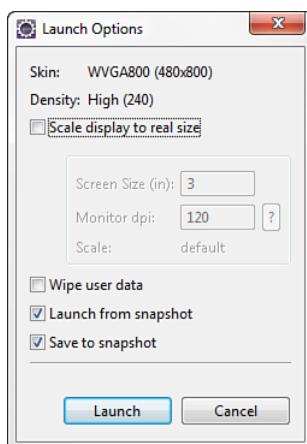


Figure 3.7 Configuring AVD launch options.

4. If necessary, swipe the screen from left to right to unlock the emulator, as shown in Figure 3.8.
5. The Snake application starts and you can play the game, as shown in Figure 3.9.

You can interact with the Snake application through the emulator and play the game. You can also launch the Snake application from the Application drawer at any time by clicking its application icon. There is no need to shut down and restart the emulator every time you rebuild and reinstall your application for testing. Simply leave the emulator running on your computer in the background while you work in Eclipse and then redeploy using the Debug configuration again.

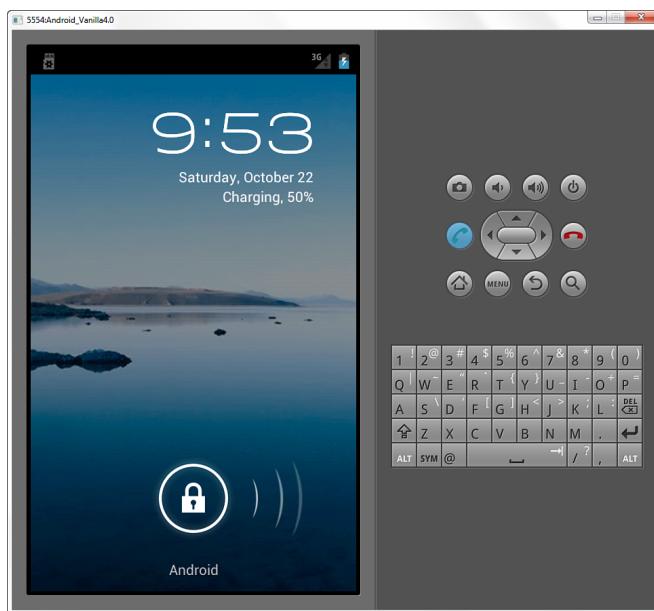


Figure 3.8 The Android emulator launching (locked).

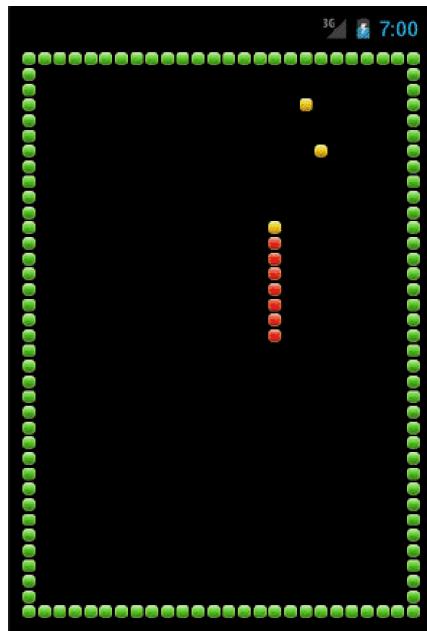


Figure 3.9 The Snake game in the Android emulator.

## Building Your First Android Application

Now it's time to write your first Android application from scratch. To get your feet wet, you will start with a simple "Hello World" application and build upon it to explore some of the features of the Android platform in more detail.



### Tip

The code examples provided in this chapter are taken from the MyFirstAndroidApp application. The source code for the MyFirstAndroidApp application is provided for download on the book's websites.

## Creating and Configuring a New Android Project

You can create a new Android application in much the same way as when you added the Snake application to your Eclipse workspace.

The first thing you need to do is create a new project in your Eclipse workspace. The Android Project Wizard creates all the required files for an Android application. Follow these steps within Eclipse to create a new project:

1. Choose File, New, Android Project, or choose the Android Project creator icon, which looks like a folder () on the Eclipse toolbar.
2. Choose a project name, as shown in Figure 3.10. In this case, name the project MyFirstAndroidApp.

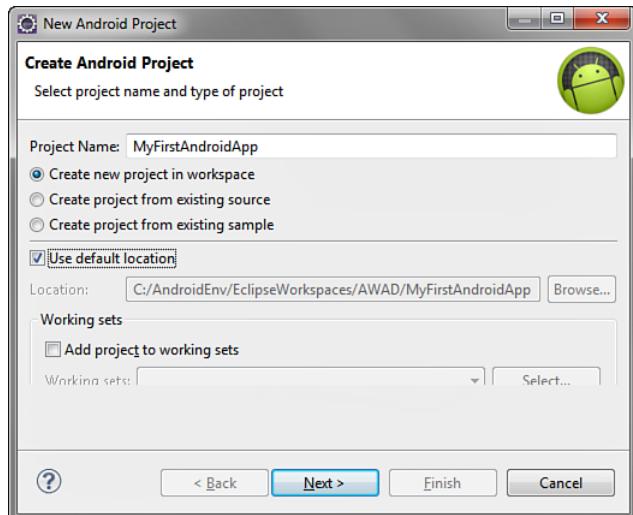


Figure 3.10 Configuring a new Android project.

3. Choose a location for the project files. Because this is a new project, select the Create New Project in Workspace radio button. Check the Use Default Location check box or change the directory to wherever you want to store the source files. Click Next.
4. Select a build target for your application, as shown in Figure 3.11. Choose a target that is compatible with the Android devices you have in your possession. For this example, you might use the Android 2.3.3 target or, for Ice Cream Sandwich devices, Android 4.0 (API Level 14). Click Next.

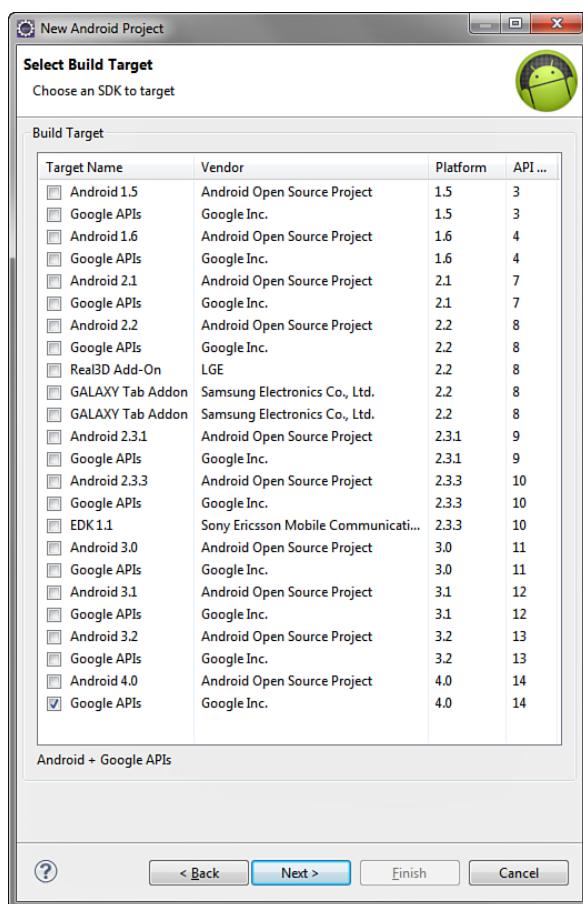


Figure 3.11 Choosing a build target for a new Android project.

5. Configure your application information. Choose an application name. The application name is the “friendly” name of the application and the name shown with the icon on the application launcher. In this case, the application name is “My First Android App.”
6. Choose a package name. Here you should follow standard package namespace conventions for Java. Because all our code examples in this book fall under the `com.androidbook.*` namespace, we will use the package name `com.androidbook.myfirstandroidapp`, but you are free to choose your own package name.
7. Check the Create Activity check box. This instructs the wizard to create a default launch activity for the application. Call this Activity class `MyFirstAndroidAppActivity`.
8. Set the minimum SDK version. This value should be the same or lower than the target SDK API level. Because our application will be compatible with just about any Android device, you can set this number low (like to 4 to represent Android 1.6) or at the target API level to avoid annoying warnings in Eclipse. Make sure you set the minimum SDK version to encompass any test devices you have available so you can successfully install the application on them.

Your project settings should look like Figure 3.12.

9. Finally, click the Finish button.

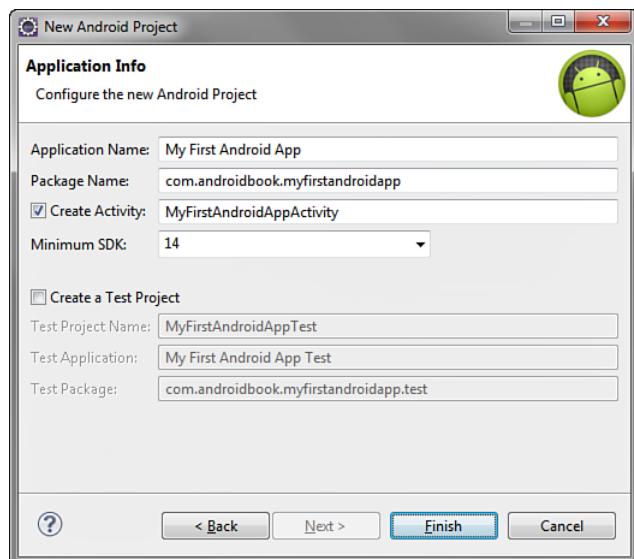


Figure 3.12 Configuring My First Android App using the Android Project Wizard.

## Core Files and Directories of the Android Application

Every Android application has a set of core files that are created and used to define the functionality of the application. The following files are created by default with a new Android application:

- **AndroidManifest.xml**—The central configuration file for the application. It defines your application's capabilities and permissions as well as how it runs.
- **project.properties**—A generated build file used by Eclipse and the Android ADT plug-in. It defines your application's build target and other build system options, as required. Do not edit this file.
- **proguard.cfg**—A generated build file used by Eclipse, ProGuard, and the Android ADT plug-in. Edit this file to configure your code optimization and obfuscation settings for release builds.
- **/src folder**—Required folder for all source code.
- **/src/com/androidbook/myfirstandroidapp/MyFirstAndroidAppActivity.java**—Main entry point to this application, named `MyFirstAndroidAppActivity`. This activity has been defined as the default launch activity in the Android manifest file.
- **/gen/com/androidbook/myfirstandroidapp/R.java**—A generated resource management source file. Do not edit this file.
- **/assets folder**—Required folder where uncompiled file resources can be included in the project. Application assets are pieces of application data (files, directories) that you do not want managed as application resources.
- **/res folder**—Required folder where all application resources are managed. Application resources include animations, drawable graphics, layout files, data-like strings and numbers, and raw files.
- **/res/drawable-\***—Application icon graphic resources are included in several sizes for different device screen resolutions.
- **/res/layout/main.xml**—Layout resource file used by `MyFirstAndroidAppActivity` to organize controls on the main application screen.
- **/res/values/strings.xml**—The resource file where string resources are defined.

A number of other files are saved on disk as part of the Eclipse project in the workspace. However, the files and resource directories included in the list here are the important project files you will use on a regular basis.

## Creating an AVD for Your Project

The next step is to create an AVD that describes what type of device you want to emulate when running the application. For this example, we can use the AVD we created for

the Snake application. An AVD describes a device, not an application. Therefore, you can use the same AVD for multiple applications. You can also create similar AVDs with the same configuration but different data (such as different applications installed and different SD card contents).

## Creating a Launch Configuration for Your Project

Next, you must create a Run and Debug launch configuration in Eclipse to configure the circumstances under which the MyFirstAndroidApp application builds and launches. The launch configuration is where you configure the emulator options to use and the entry point for your application.

You can create Run configurations and Debug configurations separately, with different options for each. Begin by creating a Run configuration for the application. Follow these steps to create a basic Run configuration for the MyFirstAndroidApp application:

1. Choose Run, Run Configurations (or right-click the project and choose Run As).
2. Double-click Android Application.
3. Name your Run configuration `MyFirstAndroidAppRunConfig`.
4. Choose the project by clicking the Browse button and choosing the MyFirstAndroidApp project.
5. Switch to the Target tab and set the Device Target Selection Mode to Manual.



### Tip

If you leave the Device Target Selection Mode set to Automatic when you choose Run or Debug in Eclipse, your application is automatically installed and run on the device if the device is plugged in. Otherwise, the application starts in the emulator with the specified AVD. By choosing Manual, you are always prompted for whether (a) you want your application to be launched in an existing emulator; (b) you want your application to be launched in a new emulator instance and are allowed to specify an AVD; or (c) you want your application to be launched on the device (if it's plugged in). If any emulator is already running, the device is then plugged in, and the mode is set to Automatic, you see this same prompt, too.

Now create a Debug configuration for the application. This process is similar to creating a Run configuration. Follow these steps to create a basic Debug configuration for the MyFirstAndroidApp application:

1. Choose Run, Debug Configurations (or right-click the project and choose Debug As).
2. Double-click Android Application.
3. Name your Debug configuration `MyFirstAndroidAppDebugConfig`.

4. Choose the project by clicking the Browse button and choosing the MyFirstAndroidApp project.
5. Switch to the Target tab and set the Device Target Selection Mode to Manual.
6. Click Apply and then click Close.

You now have a Debug configuration for your application.

## Running Your Android Application in the Emulator

Now you can run the MyFirstAndroidApp application using the following steps:

1. Choose the Run As icon drop-down menu on the toolbar ().
2. Pull the drop-down menu and choose the Run configuration you created. (If you do not see it listed, choose the Run Configurations... item and select the appropriate configuration. The Run configuration shows up on this drop-down list the next time you run the configuration.)
3. Because you chose the Manual Target Selection mode, you are now prompted for your emulator instance. Change the selection to Launch a New Android Virtual Device and then select the AVD you created, as shown in Figure 3.13. Here you can choose from an already-running emulator or launch a new instance with an AVD that is compatible with the application settings.

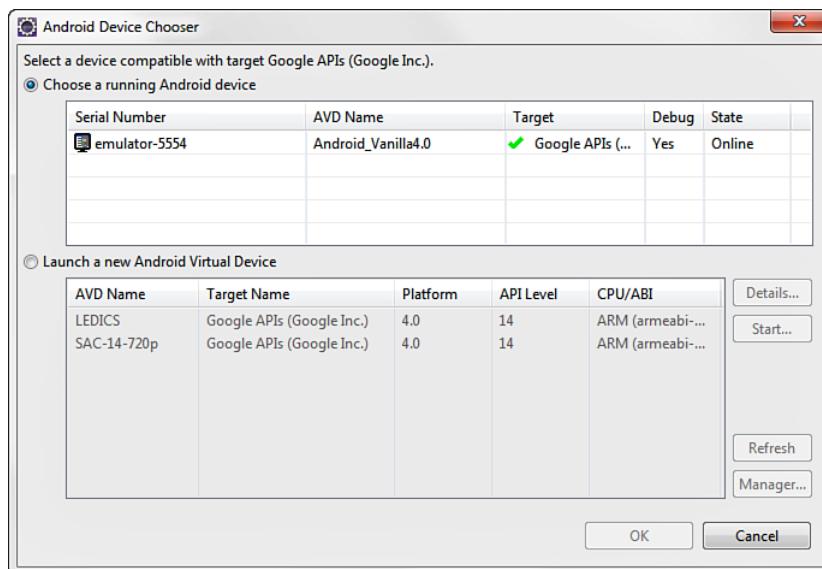


Figure 3.13 Manually choosing a target selection mode.

4. The Android emulator starts up, which might take a moment.
5. Click the Menu button or push the slider to the right to unlock the emulator.
6. The application starts, as shown in Figure 3.14.



Figure 3.14 My First Android App running in the emulator.

7. Click the Back button in the Emulator to end the game or click Home to suspend it.
8. Click the grid button to browse all installed applications. Your screen looks something like Figure 3.15.
9. Click the My First Android Application icon to launch the application again.

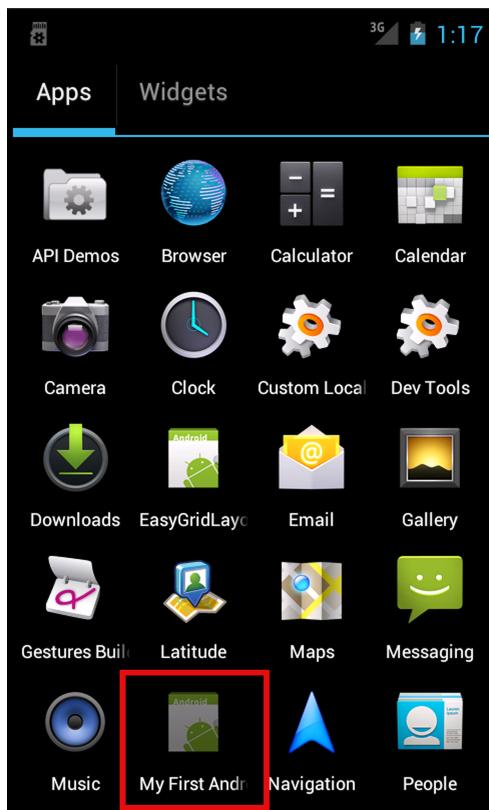


Figure 3.15 The My First Android App icon shown in the application listing.

## Debugging Your Android Application in the Emulator

Before we go any further, you need to become familiar with debugging in the emulator. To illustrate some useful debugging tools, let's manufacture an error in the My First Android Application.

In your project, edit the source file called `MyFirstAndroidApp.java`. Create a new method called `forceError()` in your class and make a call to this method in your `Activity` class's `onCreate()` method. The `forceError()` method forces a new unhandled error in your application.

The `forceError()` method should look something like this:

```
public void forceError() {  
    if(true) {  
        throw new Error("Whoops");  
    }  
}
```

It's probably helpful at this point to run the application and watch what happens. Do this using the Run configuration first. In the emulator, you see that the application has stopped unexpectedly. You are prompted by a dialog that enables you to forcefully close the application, as shown in Figure 3.16.

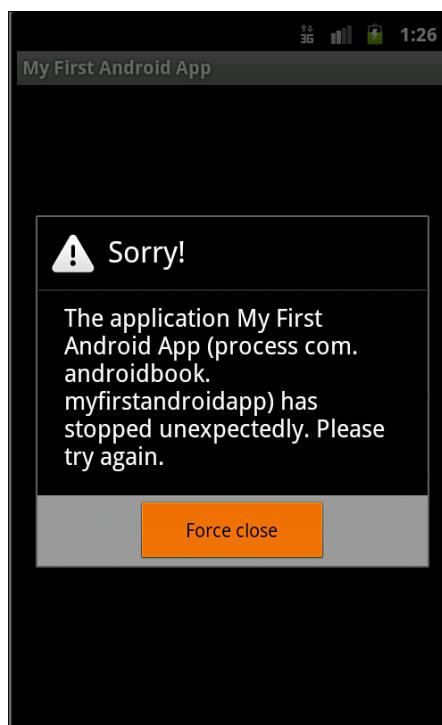


Figure 3.16 My First Android App crashing gracefully.

Shut down the application but keep the emulator running. Now it's time to debug. You can debug the `MyFirstAndroidApp` application using the following steps:

1. Choose the Debug As icon drop-down menu on the toolbar.
2. Pull the drop-down menu and choose the Debug configuration you created.  
(If you do not see it listed, choose the Debug Configurations... item and select the

appropriate configuration. The Debug configuration shows up on this drop-down list the next time you run the configuration.)

3. Continue as you did with the Run configuration and choose the appropriate AVD and then launch the emulator again, unlocking it if needed.

It takes a moment for the debugger to attach. If this is the first time you've debugged an Android application, you may need to click through some dialogs, such as the one shown in Figure 3.17, the first time your application attaches to the debugger.

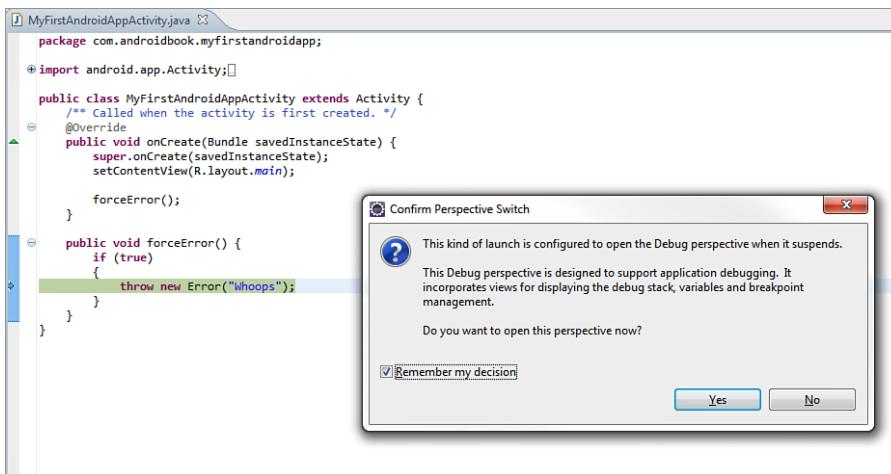


Figure 3.17 Switching debug perspectives for Android emulator debugging.

In Eclipse, use the Debug perspective to set breakpoints, step through code, and watch the LogCat logging information about your application. This time, when the application fails, you can determine the cause using the debugger. You might need to click through several dialogs as you set up to debug within Eclipse. If you allow the application to continue after throwing the exception, you can examine the results in the Debug perspective of Eclipse. If you examine the LogCat logging pane, you see that your application was forced to exit due to an unhandled exception (see Figure 3.18).

Specifically, there's a red `AndroidRuntime` error: `java.lang.Error: Whoops`. Back in the emulator, click the Force Close button. Now set a breakpoint on the `forceError()` method by right-clicking the left side of the line of code and choosing Toggle Breakpoint (or pressing `Ctrl+Shift+B`).

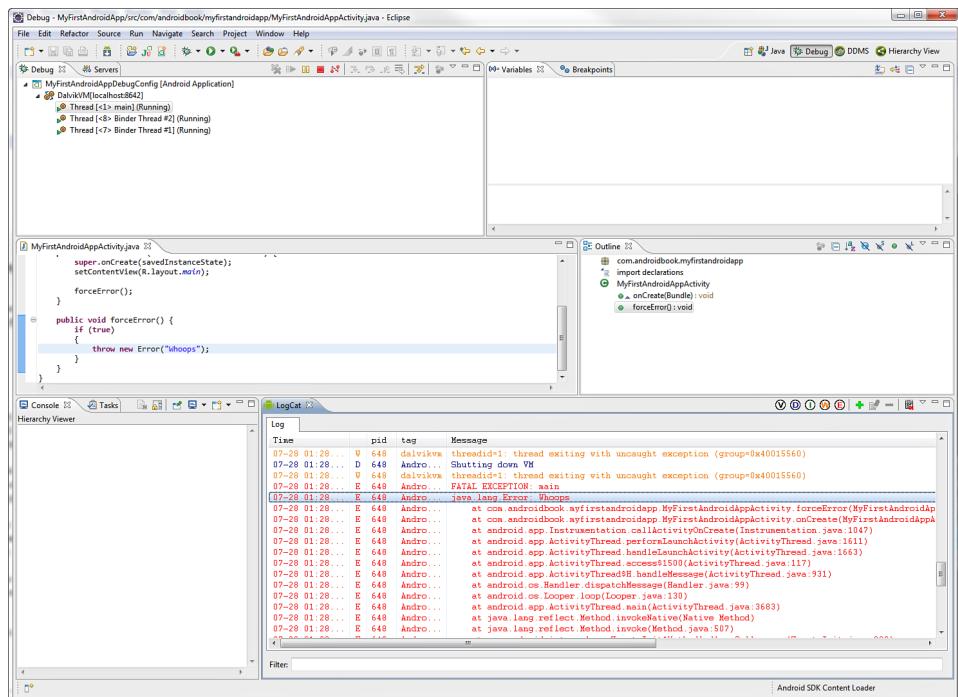


Figure 3.18 Debugging My First Android App in Eclipse.



### Tip

In Eclipse, you can step through code using Step Into (F5), Step Over (F6), Step Return (F7), or Resume (F8). On Mac OS X, you might find that the F8 key is mapped globally. If you want to use the keyboard convenience command, you might want to change the keyboard mapping in Eclipse by choosing Eclipse, Preferences, General, Keys and then finding the entry for Resume and changing it to something else. Alternatively, you can change the Mac OS X global mapping by going to System Preferences, Keyboard & Mouse, Keyboard Shortcuts and then changing the mapping for F8 to something else.

In the emulator, restart your application and step through your code. You see that your application has thrown the exception and then the exception shows up in the Variable Browser pane of the Debug Perspective. Expanding its contents shows that it is the “Whoops” error.

This is a great time to crash your application repeatedly and get used to the controls. While you’re at it, switch over to the DDMS perspective. You note the emulator has a list of processes running on the device, such as `system_process` and `com.android.phone`. If you launch `MyFirstAndroidApp`, you see `com.android.book.myfirstandroidapp` show up as a process on the emulator listing. Force the app

to close because it crashes, and note that it disappears from the process list. You can use DDMS to kill processes, inspect threads and the heap, and access the phone file system.

## Adding Logging Support to Your Android Application

Before you start diving into the various features of the Android SDK, you should familiarize yourself with logging, a valuable resource for debugging and learning Android.

Android logging features are in the `Log` class of the `android.util` package. Some helpful methods in the `android.util.Log` class are shown in Table 3.1.

Table 3.1 Commonly Used Logging Methods

Method	Purpose
<code>Log.e()</code>	Log errors
<code>Log.w()</code>	Log warnings
<code>Log.i()</code>	Log informational messages
<code>Log.d()</code>	Log debug messages
<code>Log.v()</code>	Log verbose messages

To add logging support to `MyFirstAndroidApp`, edit the file `MyFirstAndroidApp.java`.

First, you must add the appropriate import statement for the `Log` class:

```
import android.util.Log;
```

### Tip

To save time in Eclipse, you can use the imported classes in your code and add the imports needed by hovering over the imported class name and choosing the Add Imported Class option.

You can also use the Organize Imports command (Ctrl+Shift+O in Windows or Command+Shift+O on a Mac) to have Eclipse automatically organize your imports. This removes unused imports and adds new ones for packages used but not imported. If a naming conflict arises, as it often does with the `Log` class, you can choose the package you intended to use.

Next, within the `MyFirstAndroidApp` class, declare a constant string that you use to tag all logging messages from this class. You can use the LogCat utility within Eclipse to filter your logging messages to this `DEBUG_TAG` tag string:

```
private static final String DEBUG_TAG= "MyFirstAppLogging";
```

Now, within the `onCreate()` method, you can log something informational:

```
Log.i(DEBUG_TAG,  
"In the onCreate() method of the MyFirstAndroidAppActivity Class");
```

While you're here, you must comment out your previous `forceError()` call so that your application doesn't fail. Now you're ready to run `MyFirstAndroidApp`. Save your work and debug it in the emulator. You notice that your logging messages appear in the LogCat listing, with the Tag field `MyFirstAppLogging` (see Figure 3.19).

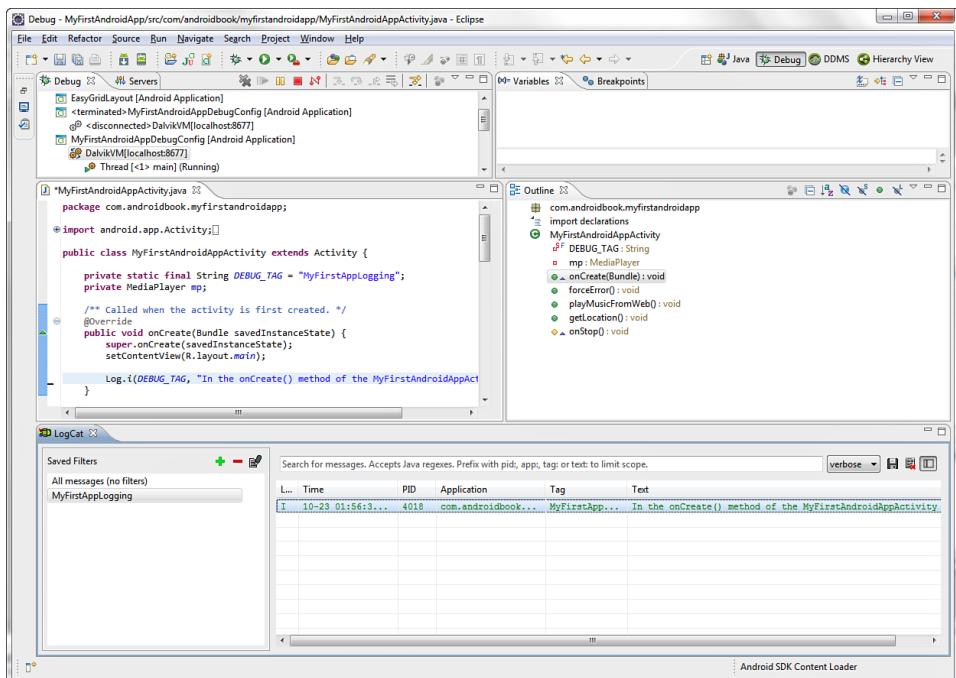


Figure 3.19 A LogCat log for My First Android App.

## Adding Some Media Support to Your Application

Next, let's add some pizzazz to `MyFirstAndroidApp` by having the application play an MP3 music file. Android media player features are found in the `MediaPlayer` class of the `android.media` package.

You can create `MediaPlayer` objects from existing application resources or by specifying a target file using a Uniform Resource Identifier (URI). For simplicity, we begin by accessing an MP3 using the `Uri` class from the `android.net` package.

Some methods in the `android.media.MediaPlayer` and `android.net.Uri` classes are shown in Table 3.2.

Table 3.2 Commonly Used MediaPlayer and Uri Parsing Methods

Method	Purpose
<code>MediaPlayer.create()</code>	Creates a new media player with a given target to play
<code>MediaPlayer.start()</code>	Starts media playback
<code>MediaPlayer.stop()</code>	Stops media playback
<code>MediaPlayer.release()</code>	Releases the media player resources
<code>Uri.parse()</code>	Instantiates a <code>Uri</code> object from an appropriately formatted URI address

To add MP3 playback support to `MyFirstAndroidApp`, edit the file `MyFirstAndroidApp.java`. First, you must add the appropriate import statements for the `MediaPlayer` class:

```
import android.media.MediaPlayer;
import android.net.Uri;
```

Next, within the `MyFirstAndroidApp` class, declare a member variable for your `MediaPlayer` object:

```
private MediaPlayer mp;
```

Now, create a new method called `playMusicFromWeb()` in your class and make a call to this method in your `onCreate()` method. The `playMusicFromWeb()` method creates a valid `Uri` object, creates a `MediaPlayer` object, and starts the MP3 playing. If the operation should fail for some reason, the method logs a custom error with your logging tag. The `playMusicFromWeb()` method should look something like this:

```
public void playMusicFromWeb() {
    try {
        Uri file = Uri.parse("http://www.perlgurl.org/podcast/archives"
            + "/podcasts/PerlgurlPromo.mp3");
        mp = MediaPlayer.create(this, file);
        mp.start();
    }
    catch (Exception e) {
        Log.e(DEBUG_TAG, "Player failed", e);
    }
}
```

As of Android 4.0 (API Level 14), using the `MediaPlayer` class to access media content on the Web requires the `INTERNET` permission to be registered in the application's Android manifest file. Finally, your application requires special permissions to access location-based functionality. You must register this permission in your `AndroidManifest.xml` file. To add permissions to your application, perform the following steps:

1. Double-click the `AndroidManifest.xml` file.
2. Switch to the Permissions tab.
3. Click the Add button and choose Uses Permission.
4. In the right pane, select `android.permission.INTERNET`.
5. Save the file.

Later on, you'll learn all about the various `Activity` states and callbacks that could contain portions of the `playMusicFromWeb()` method. For now, know that the `onCreate()` method is called every time the user navigates to the `Activity` (forward or backward) and whenever he or she rotates the screen or causes other device configuration changes. This doesn't cover all cases, but will work well enough for this example.

And finally, you want to cleanly exit when the application shuts down. To do this, you need to override the `onStop()` method of your `Activity` class and stop the `MediaPlayer` object and release its resources. The `onStop()` method should look something like this:

```
protected void onStop() {  
    if (mp != null) {  
        mp.stop();  
        mp.release();  
    }  
    super.onStop();  
}
```



### Tip

In Eclipse, you can right-click within the class and choose Source (or press Alt+Shift+S). Choose the option Override/Implement Methods and select the `onStop()` method.

Now, if you run `MyFirstAndroidApp` in the emulator (and you have an Internet connection to grab the data found at the URI location), your application plays the MP3. When you shut down the application, the `MediaPlayer` is stopped and released appropriately.

### Adding Location-Based Services to Your Application

Your application knows how to say “Hello” and play some music, but it doesn’t know where it’s located. Now is a good time to become familiar with some simple location-based calls to get the GPS coordinates. To have some fun with location-based services and maps integration, you will use some of the Google applications available on typical Android devices—specifically, the Maps application. You do not need to create another AVD, because you included the Google APIs as part of the target for the AVD you already created.

## Configuring the Location of the Emulator

The emulator does not have location sensors, so the first thing you need to do is seed your emulator with some GPS coordinates. You can find the exact steps for how to do this in Appendix A, in the section “Configuring the GPS Location of the Emulator.”

After you have configured the location of your emulator, the Maps application should now display your simulated location, as shown in Figure 3.20.

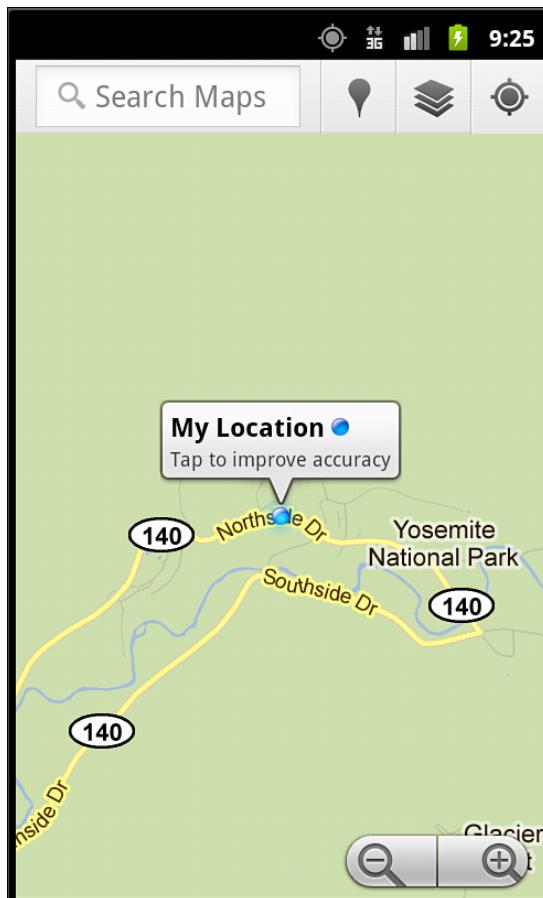


Figure 3.20 Setting the location of the emulator to Yosemite Valley.

Your emulator now has a simulated location: Yosemite Valley!

## Finding the Last Known Location

To add location support to `MyFirstAndroidApp`, edit the file `MyFirstAndroidApp.java`. First, you must add the appropriate import statements:

```
import android.location.Location;
import android.location.LocationManager;
```

Now, create a new method called `getLocation()` in your class and make a call to this method in your `onCreate()` method. The `getLocation()` method gets the last known location on the device and logs it as an informational message. If the operation fails for some reason, the method logs an error.

The `getLocation()` method should look something like this:

```
public void getLocation() {
    try {
        LocationManager locMgr = (LocationManager)
            getSystemService(LOCATION_SERVICE);
        Location recentLoc = locMgr.
            getLastKnownLocation(LocationManager.GPS_PROVIDER);
        Log.i(DEBUG_TAG, "loc: " + recentLoc.toString());
    }
    catch (Exception e) {
        Log.e(DEBUG_TAG, "Location failed", e);
    }
}
```

Finally, your application requires special permissions to access location-based functionality. You must register this permission in your `AndroidManifest.xml` file. To add location-based service permissions to your application, perform the following steps:

1. Double-click the `AndroidManifest.xml` file.
2. Switch to the Permissions tab.
3. Click the Add button and choose Uses Permission.
4. In the right pane, select `android.permission.ACCESS_FINE_LOCATION`.
5. Save the file.

Now, if you run My First Android App in the emulator, your application logs the GPS coordinates you provided to the emulator as an informational message, viewable in the LogCat pane of Eclipse.

## Debugging Your Application on the Hardware

You mastered running applications in the emulator. Now let's put the application on real hardware. First, you must register your application as debuggable in your `AndroidManifest.xml` file. To do this, perform the following steps:

1. Double-click the `AndroidManifest.xml` file.
2. Change to the Application tab.
3. Set the Debuggable application attribute to true.
4. Save the file.

You can also modify the `application` element of the `AndroidManifest.xml` file directly with the `android:debuggable` attribute, as shown here:

```
<application ... android:debuggable="true">
```

Now, connect an Android device to your computer via USB and re-launch the Run configuration or Debug configuration of the application. Because you chose Manual mode for the configuration, you should now see a real Android device listed as an option in the Android Device Chooser (see Figure 3.21).

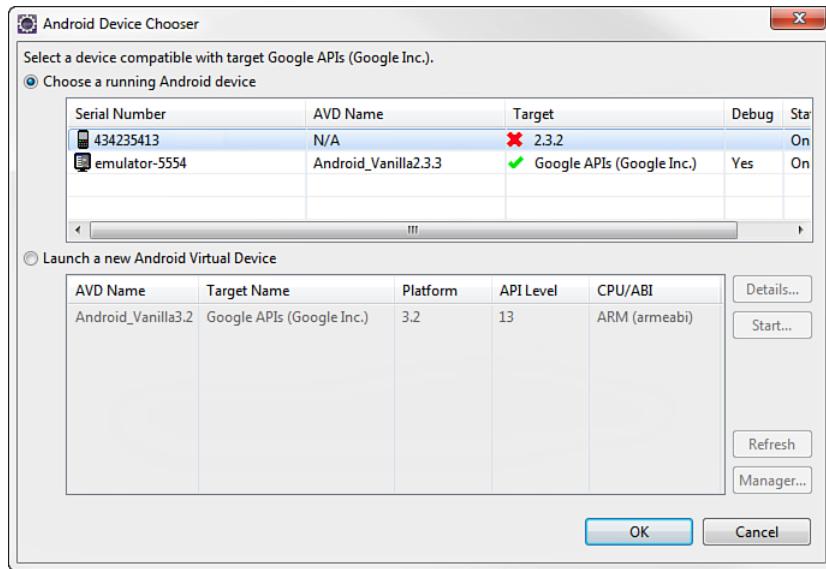


Figure 3.21 Android Device Chooser with USB-connected Android device.

Choose the Android device as your target, and you see that the My First Android App application gets loaded onto the Android device and launched, just as before. Provided you have enabled the development debugging options on the device, you can debug the application here as well. You can tell the device is actively using a USB debugging connection, because a little Android bug-like icon appears in the notification bar ((Notification icon)). Figure 3.22 shows a screenshot of the application running on a real device (in this case, a smartphone running Android 2.3.2).

Debugging on the device is much the same as debugging on the emulator, but with a couple of exceptions. You cannot use the emulator controls to do things such as send an SMS or configure the location to the device, but you can perform real actions (true SMS, actual location data) instead.

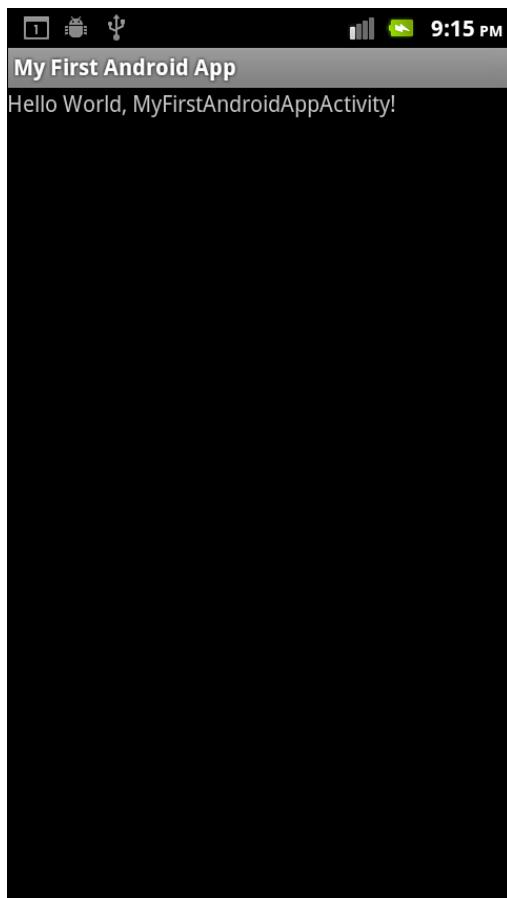


Figure 3.22 My First Android App running on Android device hardware.

## Summary

This chapter showed you how to add, build, run, and debug Android projects using Eclipse. You started by testing your development environment using a sample application from the Android SDK and then you created a new Android application from scratch using Eclipse. You also learned how to make some quick modifications to the application, demonstrating some exciting Android features you learn about in future chapters.

In the next few chapters, you learn about the tools available for use developing Android applications and then focus on the finer points about defining your Android

application using the application manifest file and how the application lifecycle works. You also learn how to organize your application resources, such as images and strings, for use within your application.

## References and More Information

Android SDK Reference regarding the application `Activity` class:

<http://d.android.com/reference/android/app/Activity.html>

Android SDK Reference regarding the application `Log` class:

<http://d.android.com/reference/android/util/Log.html>

Android SDK Reference regarding the application `MediaPlayer` class:

<http://d.android.com/reference/android/media/MediaPlayer.html>

Android SDK Reference regarding the application `Uri` class:

<http://d.android.com/reference/android/net/Uri.html>

Android SDK Reference regarding the application `LocationManager` class:

<http://d.android.com/reference/android/location/LocationManager.html>

Android Dev Guide: “Developing on a Device”:

<http://d.android.com/guide/developing/device.html>

Android Resources: “Common Tasks and How to Do Them in Android”:

<http://d.android.com/resources/faq/commontasks.html>

Android Sample Code: “Snake”:

<http://d.android.com/resources/samples/Snake>

*This page intentionally left blank*

# Mastering the Android Development Tools

Android developers are fortunate to have many tools at their disposal to help facilitate the design and development of quality applications. Some of the Android development tools are integrated into Eclipse using the ADT plug-in, whereas others must be used from the command line. In this chapter, we walk through a number of the most important tools available for use with Android. This information will help you develop Android applications faster and with fewer roadblocks.



## Note

The Android development tools are updated frequently. We have made every attempt to provide the latest steps for the latest tools. However, these steps and the user interfaces described in this chapter may change at any time. Please review the Android development website (<http://d.android.com/guide/developing/tools/>) and our book website (<http://androidbook.blogspot.com>) for the latest information.

## Using the Android Documentation

Although it is not a tool, per se, the Android documentation is a key resource for Android developers. An HTML version of the Android documentation is provided in the `docs` subfolder of the Android SDK documentation, and this should always be your first stop when you encounter a problem. You can also access the latest help documentation online at the Android Developer website, <http://developer.android.com> (or <http://d.android.com> for short). The Android documentation is organized and searchable. It is divided into seven sections:

- **Home:** This tab provides some high-level news items for Android developers, including announcements of new platform versions. You'll also find quick links for downloading the latest Android SDK, publishing your applications on the Android Market, and other helpful information.

- **SDK:** This tab provides important information about the SDK version installed on your machine. One of the most important features of this tab is the release notes, which describe any known issues for the specific installation. This information is also useful if the online help has been upgraded but you want to develop to an older version of the SDK.
- **Dev Guide:** This tab links to the Android Developer's Guide, which includes a number of FAQs, best practice guides, and a useful glossary of Android terminology for those new to the platform. The appendix section of the Dev Guide tab also details all Android platform versions (API levels), supported media formats, and lists of intents.
- **Reference:** This tab includes a searchable package and class index of all Android APIs provided as part of the Android SDK, in a Javadoc-style format. You will spend most of your time on this tab, looking up Java class documentation, checking method parameters, and other similar tasks.
- **Resources:** This tab includes links to articles, tutorials, and sample code, as well as acts as a gateway to the Android developer forums. There are a number of Google groups you can join, depending on your interests.
- **Videos:** This tab, which is available online only, is your resource for Android training videos. Here, you'll find videos about the Android platform, developer tips, and the Google I/O conference sessions.
- **Blog:** This tab links to the official Android developer blog. Check here for the latest news about the Android platform. This is the place to find how-to examples, learn how to optimize Android applications, and hear about new SDK releases and features as well as Android best practices from the designers of the platform.

Figure 4.1 shows a screenshot of the Android SDK Reference tab of the website.

Now is a good time to get to know your way around the Android SDK documentation. First, check out the online documentation and then try the local documentation.



### Tip

Different features of the Android SDK are applicable to different versions of the platform. New APIs, classes, interfaces, and methods have been introduced over time. Therefore, each item in the documentation is tagged with the API level when it was first introduced. To see whether a specific item is available in a specific platform version, check its API level, usually listed along the right side of the documentation. You can also filter the documentation to a specific API level, so that it only displays SDK features available for that platform version (see Figure 4.1, top right).

Keep in mind that this book is designed to be a companion guide in your journey to mastering Android development. It covers Android fundamentals and tries to distill a lot of information using an easily digestible format to get you up and running quickly. It then provides you with a thorough understanding of what is available and feasible on the

Android platform. It is not an exhaustive SDK reference, but a guide to best practices. You'll need to become intimately familiar with the Android SDK Java class documentation in order to be successful at designing and developing Android applications in the long run.

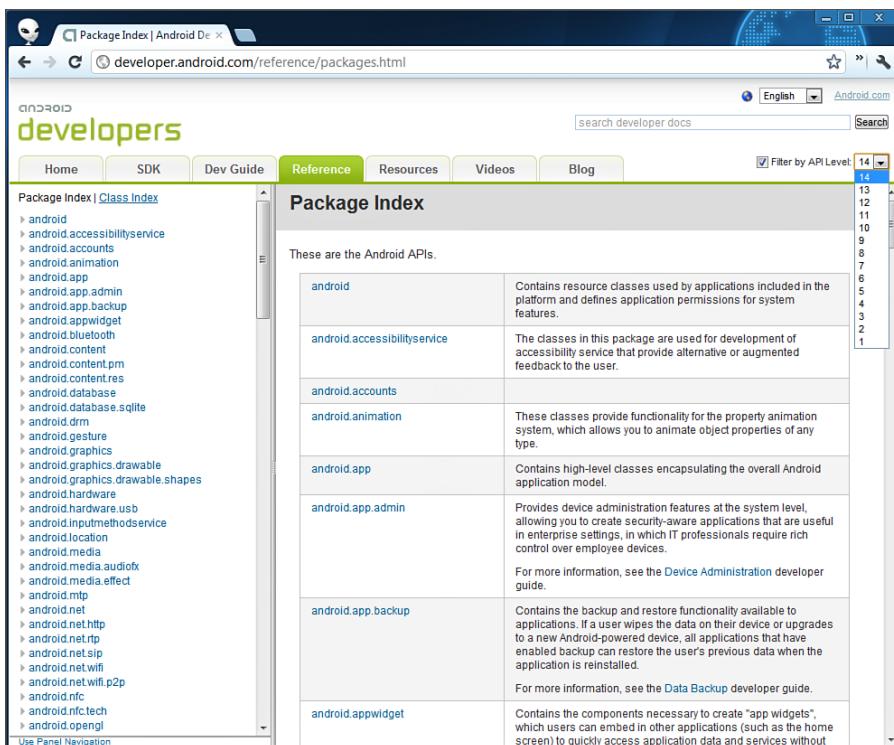


Figure 4.1 The Android Developer website.

## Leveraging the Android Emulator

Although we introduced the Android emulator as a core tool in Chapter 2, “Setting Up Your Android Development Environment,” it’s worth mentioning again. The Android emulator is probably the most powerful tool at a developer’s disposal, along with the Android SDK and Android Virtual Device Managers (which we also talked about fairly extensively in the previous two chapters). It is important for developers to learn to use the emulator and understand its limitations. The Android emulator is integrated with Eclipse, using the ADT plug-in for the Eclipse IDE. For more information about the emulator, please check out Appendix A, “The Android Emulator Quick-Start Guide.”

We suggest that you review this detailed appendix after you have looked over the other materials covered in this chapter.

Figure 4.2 shows a sample application called PeakBagger 1.0 and what it looks like when running in the Android emulator.



Figure 4.2 The Android emulator in action running the PeakBagger 1.0 application.

You can also find exhaustive information about the emulator in the Android Developer website: <http://d.android.com/guide/developing/tools/emulator.html>.

## Viewing Application Log Data with LogCat

You learned how to log application information in the last chapter using the `android.util.Log` class. The log output appears in the LogCat pane of Eclipse (available in the Debug and DDMS perspectives of Eclipse). You can also interact with `logcat` directly (more on this later in the chapter).

Even when you have a great debugger, incorporating logging support into your applications is very useful. You can then monitor your application's log output, generated on either the emulator or an attached device. Log information can be invaluable for

tracking down difficult bugs and reporting application state during the development phase of a project.

Log data is categorized by severity. When you create a new class in your project, we recommend defining a unique debug tag string for that class so that you can easily track down where a log message originated. You can use this tag to filter the logging data and find only the messages you are interested in. You can use the LogCat utility from within Eclipse to filter your log messages to the debug tag string you supplied for your application. To learn how to do this, check out the “Creating Custom Log Filters” section in Appendix C, “Eclipse IDE Tips and Tricks.”

Finally, there are some performance tradeoffs to consider when it comes to logging. Excessive logging impacts device and application performance. At a minimum, debug and verbose logging should be used only for development purposes and removed prior to application publication.

## Debugging Applications with DDMS

When it comes to debugging on the emulator or device, you need to turn your attention to the Dalvik Debug Monitor Service (DDMS) tool. DDMS is a debugging utility that is integrated into Eclipse through a special Eclipse perspective. It is also available as a standalone executable in the `/tools` directory of the Android SDK installation.

The DDMS perspective in Eclipse (see Figure 4.3) provides a number of useful features for interacting with emulators and handsets and debugging applications. You use DDMS to view and manage processes and threads running on the device, view heap data, attach to processes to debug, and a variety of other tasks.

You can find out all about DDMS and how to use its features in Appendix B, “The Android DDMS Quick-Start Guide.” We suggest that you review this detailed appendix after you have looked over the other material covered in this chapter.

## Using Android Debug Bridge (ADB)

The Android Debug Bridge (ADB) is a client/server command-line tool that enables developers to debug Android code on the emulator and the device using a standard Java IDE such as Eclipse. The DDMS and the Android Development Tools Plug-In for Eclipse both use ADB to facilitate interaction between the development environment and the device (or emulator). You can find the `adb.exe` command-line tool in the `platform-tools` directory of the Android SDK.

Developers can also use ADB to interact with the device file system, install and uninstall Android applications manually, and issue shell commands. For example, the `logcat` and `sqlite3` shell commands enable you to access logging data and application databases.

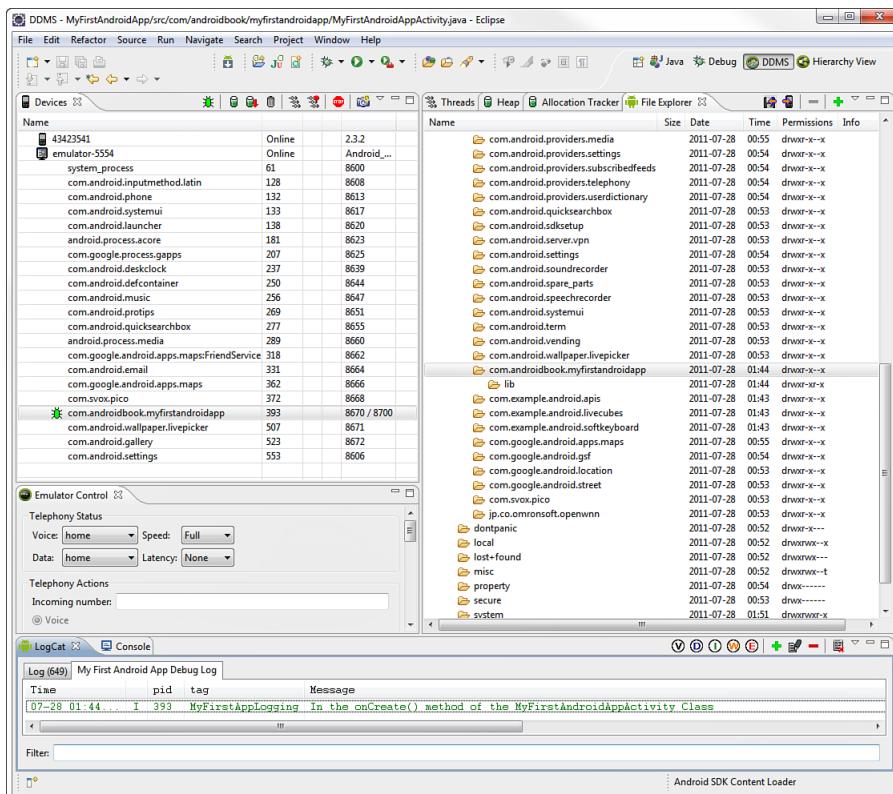


Figure 4.3 Using DDMS integrated into an Eclipse perspective.

For more information about the ADB, see the special appendix in *Android Wireless Application Development Volume II: Advanced Topics*. For an exhaustive reference, see the Android SDK Documentation at <http://d.android.com/guide/developing/tools/adb.html>.

## Using the Resource Editors and UI Designer

Eclipse is already known as a solid, well-designed development environment for Java applications. When you install the ADT plug-in for Eclipse, you add a bunch of simple Android-specific tools to help you design, develop, debug, and publish applications. Like all applications, Android apps are made up of functionality (Java code) and data (resources such as strings and graphics). The functionality is handled with the Eclipse Java editor and compiler. The ADT plug-in adds numerous special editors for creating Android-specific resource files to encapsulate application data such as strings and user interface resource templates called layouts.

Most Android resources are stored in specially formatted XML files. The resource editors and UI designer that come as part of the ADT plug-in allow you to work with application resources in a structured, graphical way, or by editing the raw XML. Two examples of resource editors include the string resource editor and the Android manifest file editor. The generic resource editor will load when you are working with XML files in the `/res` project directory hierarchy, such as string, color, or dimension resources.

The Android manifest file resource editor will load when you open the Android manifest file associated with your project. Figure 4.4 shows that the resource editor for editing Android manifest files consists of numerous tabs, which organize the contents of that resource type. Note that the last tab is always the XML tab, where you can edit the XML resource file manually, if necessary.

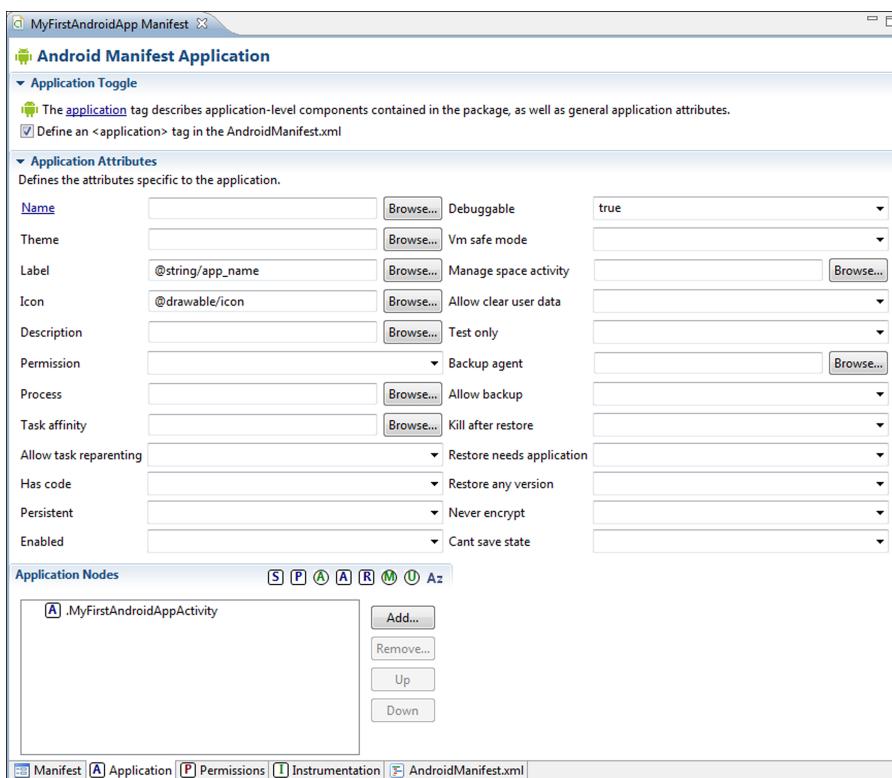


Figure 4.4 Editing the Android manifest file in Eclipse.

Android layout (user interface templates) resources are also technically XML files. However, some people prefer to be able to drag and drop controls, move them around, and preview what the user interface would look like to a real person. Recent updates to the ADT plug-in have greatly improved UI designer features for developers.

The UI designer loads whenever you open an XML file within the `/res/layout` project directory hierarchy. You can use the UI designer in Graphical Layout mode, which allows you to drag and drop controls and see what your application will look like with a variety of AVD-style configuration options (Android API level, screen resolution, orientation, theme, and more), as shown in Figure 4.5. You can also switch to XML editing mode to edit controls directly or set specific properties.

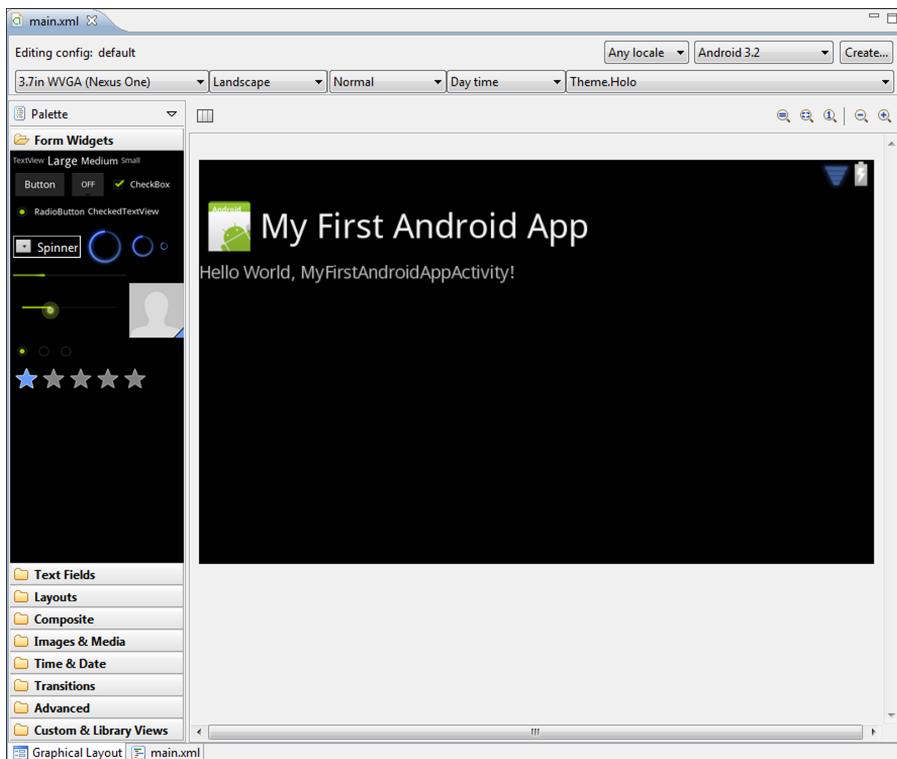


Figure 4.5 Using the UI designer in Eclipse.



### Tip

We recommend anchoring the Properties pane of Eclipse to the left of the UI designer, to edit properties of a specific selected control in a more structured manner.

We'll discuss the details of designing and developing user interfaces, as well as working with layouts and user interface controls, in Chapter 8, "Exploring User Interface Screen Elements," and Chapter 9, "Designing User Interfaces with Layouts." For now, we just

want you to be aware that your application user interface components are generally stored as resources and that the ADT plug-in for Eclipse provides some helpful tools for designing and managing these resources.

## Using the Android Hierarchy Viewer

The Android Hierarchy Viewer is a tool that identifies layout component relationships (the hierarchy) and helps developers design, debug, and profile your user interfaces.

Developers can use this tool to inspect the user interface control properties and develop pixel-perfect layouts. The Hierarchy Viewer is available as a standalone executable in the tools subdirectory of your Android SDK installation, as well as an Eclipse perspective when using the ADT plug-in for Android development. Figure 4.6 shows what the Hierarchy Viewer looks like when first launched and connected to an emulator instance, before any of the graphical views are shown. The application to be inspected has a package name called `com.mamlambo.fishbowl`.

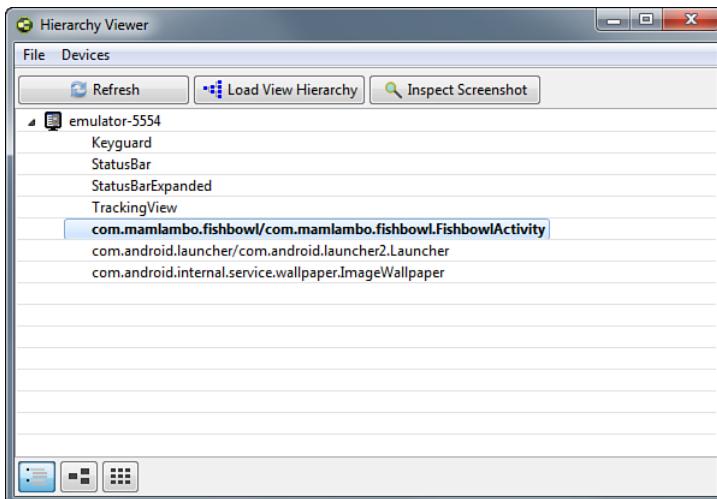


Figure 4.6 Screenshot of the Android Hierarchy Viewer standalone app when first launched.

The Hierarchy Viewer is a visual tool that can be used to inspect your application user interfaces in ways that allow you to identify and improve your layout designs. You can drill down on specific user interface controls and inspect their properties at runtime. You can save screenshots of the current application state on the emulator or the device.

The Hierarchy Viewer application is divided into two main modes:

- **Layout View Mode:** This mode shows the hierarchy of user interface controls loaded by your application in tree form. You can zoom in and select specific controls to find out lots of information about their current state, as well as profiling information to help you optimize your controls.
- **Pixel Perfect Mode:** This mode shows the user interface pixels in a zoomed-in grid fashion. This is useful for designers who need to look at very specific layout arrangements or line up views on top of images.

You can switch between the modes by using the buttons in the bottom-left corner of the tool.

## Launching the Hierarchy Viewer

To launch the Hierarchy Viewer with your application in the emulator, perform the following steps:

1. Launch your Android application in the emulator.
2. Navigate to the Android SDK tools subdirectory and launch the Hierarchy Viewer application (`hierarchyviewer.bat` on Windows), or use the Hierarchy View Eclipse perspective. We find using the standalone executable more convenient because we often want to tweak the user interface using Eclipse while we work.
3. Choose your emulator instance from the Device listing.
4. Select the application you want to view from the options available. The application must be running on that emulator to show up on the list.

## Working in Layout View Mode

The Layout View mode is invaluable for debugging drawing issues related to your application user interface controls. If you wonder why something isn't drawing correctly, try launching the Hierarchy Viewer and checking the properties for that control at runtime.



### Note

When you load an application in the Hierarchy Viewer, you will want to be aware of the fact that your application user interface does not begin at the root of the hierarchy in the tree view. In fact, there are several layers of layout controls above your application content that will appear as parent controls of your content. For example, the system status bar and title bar are higher-level controls. Your application contents are actually child controls within a `FrameLayout` control called `@+id/content`. When you load layout contents using the `setContentView()` method within your `Activity` class, you are specifying what to load within this high-level `FrameLayout`.

Figure 4.7 shows the Hierarchy Viewer loaded in Layout View mode.

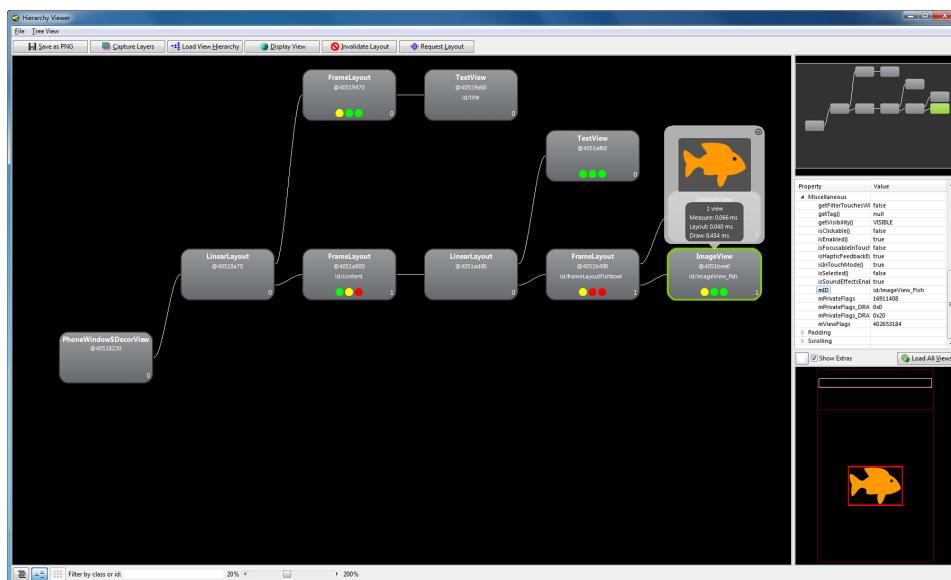


Figure 4.7 The Hierarchy Viewer tool (Layout View mode).

When you first load your application in Layout View mode, you will see several panes of information. The main pane shows the parent/child control relationships as a tree view. Each tree node represents a user interface control on the screen and shows the control's unique identifier, type, and profiling information for optimization purposes (more on this in a moment). There are also a number of smaller panes on the right side of the screen. The loupe/zoom pane allows you to quickly navigate a large tree view. The property pane shows the various properties for each tree node, when highlighted. Finally, the wire-frame model of the currently loaded user interface is displayed, with a red box highlighting the currently selected control.

### Tip

You'll have better luck navigating your application `View` objects with the Hierarchy Viewer tool if you set your `View` object `id` properties to friendly names you can remember instead of the auto-generated sequential `id` tags provided by default. For example, a `Button` control called `SubmitButton` is more descriptive than `Button01`.

You can use the Hierarchy Viewer tool to interact and debug your application user interface. Specifically, you can use the Invalidate and Request Layout features that correspond to the `View.invalidate()` and `View.requestLayout()` functions of the UI thread. These functions initiate `View` objects and draw or redraw them as necessary.

## Optimizing Your User Interface

You can also use the Hierarchy Viewer to optimize your user interface contents. You may have noticed all the little red, yellow, and green dots in tree view shown in Figure 4.7. These are performance indicators for each specific control:

- The left dot represents how long the measuring operation for this view takes.
- The middle dot represents how long the layout-rendering operation for this view takes.
- The right dot represents how long the drawing operation for this view takes.

Indicators can be red, yellow, or green and represent how each control renders in relation to other controls in the tree. They are not a strict representation of a bad or good control, per se. A red dot means that this view renders the slowest, compared to all views in your hierarchy. A yellow dot means that this view renders in the bottom 50% of all views in your hierarchy. A green dot means that this view renders in the top 50% of all views in your hierarchy. When you click a specific view within the tree, you will also see the actual performance times upon which these indicators are based.



### Tip

The Hierarchy Viewer provides control-level precision profiling. But it won't tell you if your user interface layouts are organized in the most efficient way. For that, you'll want to check out the `layoutopt` command-line tool available in the `tools` subdirectory of the Android SDK installation. This tool will help you identify unnecessary layout controls in your user interface, among other inefficiencies. Find out more at the Android Developer website: <http://d.android.com/guide/developing/tools/layoutopt.html>.

For more information on how to use this data to improve your Android application user interfaces, check out our online tutorial, “Android Tools: Leveraging the Hierarchy Viewer for UI Profiling” at <http://goo.gl/IM2e7>.

## Working in Pixel Perfect Mode

You can use the Pixel Perfect view to closely inspect your application user interface. You can also load PNG mockup files to overlay your user interface and adjust your application's look. You can access the Pixel Perfect view by clicking the button with the nine pixels on it at the bottom left of the Hierarchy Viewer.

Figure 4.8 illustrates how you can inspect the currently running application screen at the pixel level by using the loupe feature of this mode.

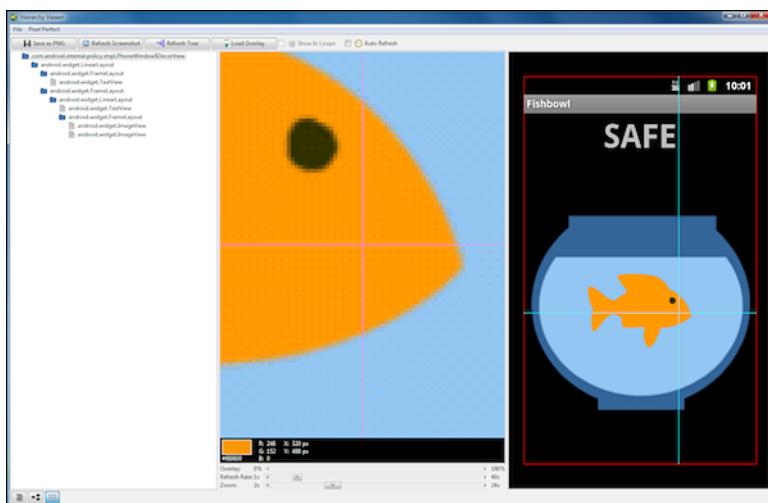


Figure 4.8 The Hierarchy Viewer tool (Pixel Perfect mode).

## Working with Nine-Patch Stretchable Graphics

Android supports Nine-Patch Stretchable Graphics, which provide flexibility for supporting different user interface characteristics, orientations, and device screens. Nine-Patch Stretchable Graphics can be created from PNG files using the `draw9patch` tool included with the `/tools` directory of the Android SDK.

Nine-Patch Stretchable Graphics are simply PNG graphics that have patches, or areas of the image, defined to scale appropriately, instead of scaling the entire image as one unit. Figure 4.9 illustrates how the image (shown as the square) is divided into nine patches. Often the center segment is transparent.

The interface for the `draw9patch` tool is straightforward. In the left pane, you can define the guides to your graphic to specify how it scales when stretched. In the right pane, you can preview how your graphic behaves when scaled with the patches you defined. Figure 4.10 shows a simple PNG file loaded in the tool, prior to its guides being set.

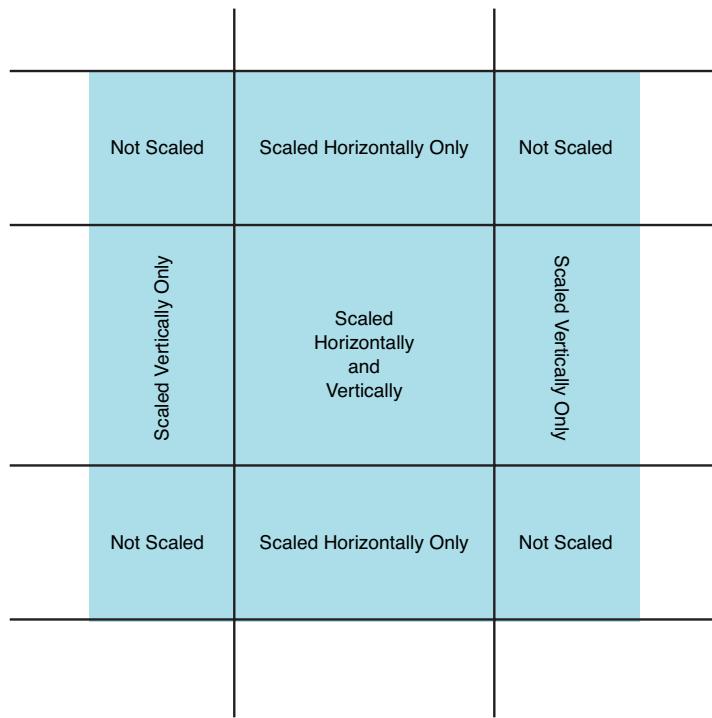


Figure 4.9 How a Nine-Patch graphic of a square is scaled.

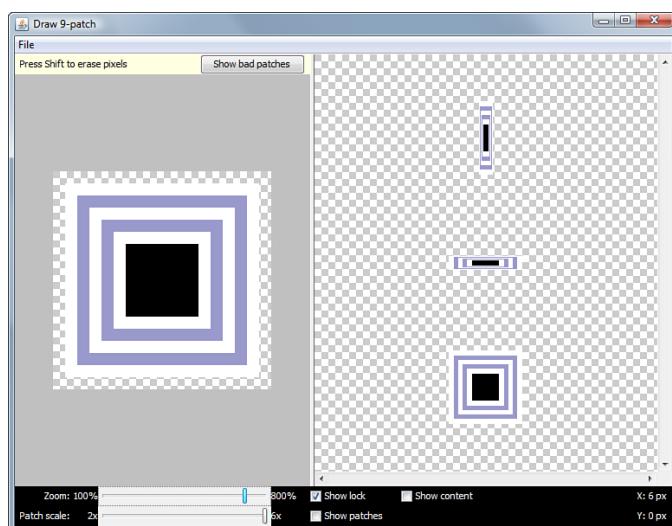


Figure 4.10 A simple PNG file before Nine-Patch processing.

To create a Nine-Patch Stretchable Graphic file from a PNG file using the draw9patch tool, perform the following steps:

1. Launch `draw9patch.bat` in your Android SDK tools subdirectory.
2. Drag a PNG file into the left pane (or use File, Open Nine-Patch).
3. Click the Show Patches check box at the bottom of the left pane.
4. Set your Patch Scale appropriately (set it higher to see more marked results).
5. Click along the left edge of your graphic to set a horizontal patch guide.
6. Click along the top edge of your graphic to set a vertical patch guide.
7. View the results in the right pane; move the patch guides until the graphic stretches as desired. Figures 4.11 and 4.12 illustrate two possible guide configurations.
8. To delete a patch guide, press Shift and click the guide pixel (black).
9. Save your graphic file. Nine-Patch graphics should end with the extension `.9.png` (for example, `little_black_box.9.png`).
10. Include your graphics file as a resource in your Android project and use just as you would a normal PNG file.

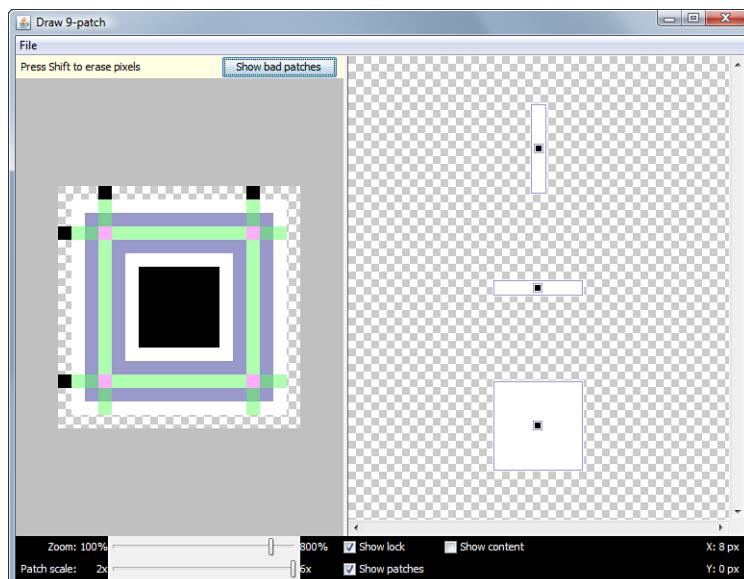


Figure 4.11 A Nine-Patch PNG file after Nine-Patch processing with some patch guides defined.

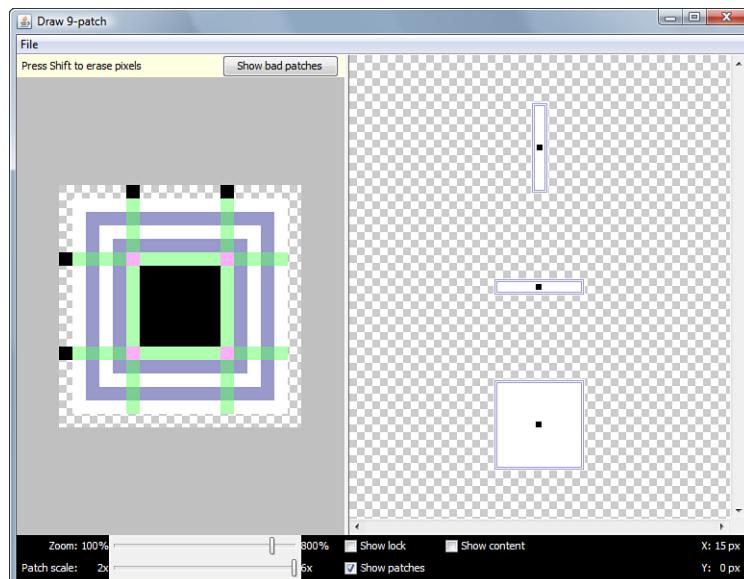


Figure 4.12 A Nine-Patch PNG file after Nine-Patch processing with different patch guides defined.

## Working with Other Android Tools

Although we've already covered the most important tools, a number of other special-purpose utilities are included with the Android SDK. Many of these tools provide the underlying functionality that has been integrated into Eclipse using the Android Development Tools (ADT) plug-in. However, if you are not using Eclipse, these tools may be used on the command line.

A complete list of the development tools that come as part of the Android SDK is available on the Android Developer website at <http://d.android.com/guide/developing/tools/index.html>. Here you'll find a description of each tool as well as a link to its official documentation. Here's a list of some useful tools we haven't yet discussed:

- **android:** This command-line tool provides much the same functionality as the Android SDK and Android Virtual Device Managers, as well as helps you create and manage projects if you are not using the Eclipse IDE with the ADT plug-in.
- **bmgr:** This shell tool is accessed through the ADB command line to interact with the Android Backup Manager. We discuss integrating backup services into your applications in *Android Wireless Application Development Volume II: Advanced Topics*.
- **dmtracedump, hprof-conv, traceview:** These tools are used for diagnostics, debug logging, and profiling of applications.

- **etc1tool:** This command-line tool lets you convert between PNG files and compressed Ericsson Texture Compression (ETC1) files. The specification for ETC1 is available at <http://goo.gl/AKV5l>.
- **logcat:** This shell tool is accessed through the ADB command line to interact with the platform logging tool, LogCat. Although you'll normally access log output through Eclipse using the ADT plug-in, you can also use this shell tool to capture, clear, and redirect log output (a useful feature if you're doing any automation, or not using Eclipse). Although the command-line logcat tool is used to provide better filters, a recent change to the Eclipse logcat view has brought this filtering power to the graphical version.
- **mksdcard:** This command-line tool lets you create SD card disk images independent of a specific AVD.
- **monkey, monkeyrunner:** These are tools you can use to test your application and implement automated testing suites. We discuss unit testing and test opportunities for your applications in Chapter 18, “Testing Android Applications.”
- **sqlite3:** This shell tool is accessed through the ADB command line to interact with the SQLite databases. We discuss using SQLite databases in your applications extensively in *Android Wireless Application Development Volume II: Advanced Topics*.
- **zipalign:** This command-line tool is used to align your APK file after it has been signed for publication. This tool is only necessary if you do not use the Eclipse Export Wizard to compile, package, sign, and align your application. We discuss these steps in Chapter 19, “Publishing Your Android Application.”

## Summary

The Android SDK ships with a number of powerful tools to help with common Android development tasks. The Android documentation is an essential reference for developers. The Android emulator can be used for running and debugging Android applications virtually, without the need for an actual device. The DDMS debugging tool, which is integrated into the Eclipse development environment as a perspective, is useful for monitoring emulators and devices. ADB is the powerful command-line tool behind many of the features of DDMS and the ADT plug-in for Eclipse. The Hierarchy Viewer and layoutopt tools can be used to design and optimize your user interface controls, whereas the Nine-Patch tool allows you to create stretchable graphics for use within your apps. There are also a number of other tools to aid developers with different development tasks, from design to development, testing, and publication.

## References and More Information

Google's Android Developer's Online SDK Reference:

<http://d.android.com/reference/packages.html>

Android Dev Guide: "Android Emulator":

<http://d.android.com/guide/developing/tools/emulator.html>

Android Dev Guide: "Dalvik Debug Monitor Server (DDMS)":

<http://d.android.com/guide/developing/debugging/ddms.html>

Android Dev Guide: "Android Debug Bridge (ADB)":

<http://d.android.com/guide/developing/tools/adb.html>

Android Dev Guide: "Draw 9-Patch":

<http://d.android.com/guide/developing/tools/draw9patch.html>

Android Dev Guide: "Debugging and Profiling User Interfaces":

<http://d.android.com/guide/developing/debugging/debugging-ui.html>



# Android Application Basics

- 5** Understanding the Anatomy of an Android Application
- 6** Defining Your Application Using the Android Manifest File
- 7** Managing Application Resources

*This page intentionally left blank*

# Understanding the Anatomy of an Android Application

**C**lassical computer science classes often define a program in terms of functionality and data, and Android applications are no different. They perform tasks, display information to the screen, and act upon data from a variety of sources.

Developing Android applications for mobile devices with limited resources requires a thorough understanding of the application lifecycle. Android uses its own terminology for these application building blocks—terms such as context, activity, and intent. This chapter familiarizes you with the most important terms, and their related Java class components, used by Android applications.

## Mastering Important Android Terminology

This chapter introduces you to the terminology used in Android application development and provides you with a more thorough understanding of how Android applications function and interact with one another. Here are some of the important terms covered in this chapter:

- **Context:** The context is the central command center for an Android application. Most application-specific functionality can be accessed or referenced through the context. The `Context` class (`android.content.Context`) is a fundamental building block of any Android application and provides access to application-wide features such as the application's private files and device resources as well as system-wide services. The application-wide `Context` object is instantiated as an `Application` object (`android.app.Application`).
- **Activity:** An Android application is a collection of tasks, each of which is called an activity. Each activity within an application has a unique task or purpose. The `Activity` class (`android.app.Activity`) is a fundamental building block of any Android application, and most applications are made up of several activities. Typically, this purpose is to handle the display of a single screen, but only thinking in terms of “an activity is a screen” is too simplistic. An `Activity` class extends the `Context` class, so it also has all of the functionality of the `Context` class.

- **Fragment:** An activity has a unique task or purpose, but it can be further componentized, each of which is called a fragment. Each fragment within an application has a unique task or purpose within its parent activity. The `Fragment` class (`android.app.Fragment`) is often used to organize activity functionality in such a way as to allow a more flexible user experience across various screen sizes, orientations, and aspect ratios. It is commonly used to hold the code and screen logic for placing the same user interface component in multiple screens, which are represented by multiple `Activity` classes.
- **Intent:** The Android operating system uses an asynchronous messaging mechanism to match task requests with the appropriate activity. Each request is packaged as an intent. You can think of each such request as a message stating an intent to *do* something. Using the `Intent` class (`android.content.Intent`) is the primary method in which application components such as activities and services communicate with one another.
- **Service:** Tasks that do not require user interaction can be encapsulated in a service. A service is most useful when the operations are lengthy (offloading time-consuming processing) or need to be done regularly (such as checking a server for new mail). Whereas activities run in the foreground and generally have a user interface, the `Service` class (`android.app.Service`) is used to handle background operations related to an Android application. The `Service` class extends the `Context` class.

## Using the Application Context

The application Context is the central location for all top-level application functionality. The `Context` class can be used to manage application-specific configuration details as well as application-wide operations and data. Use the application Context to access settings and resources shared across multiple `Activity` instances.

### Retrieving the Application Context

You can retrieve the `Context` for the current process using the `getApplicationContext()` method, found in common class such as `Activity` and `Service`, like this:

```
Context context = getApplicationContext();
```

### Using the Application Context

After you have retrieved a valid application Context object, it can be used to access application-wide features and services, including the following:

- Retrieving application resources such as strings, graphics, and XML files
- Accessing application preferences

- Managing private application files and directories
- Retrieving uncompiled application assets
- Accessing system services
- Managing a private application database (SQLite)
- Working with application permissions



### Warning

Because the `Activity` class is derived from the `Context` class, you can sometimes use this instead of retrieving the application `Context` explicitly. However, don't be tempted to just use your `Activity` `Context` in all cases because doing so can lead to memory leaks. You can find a great article on this topic at <http://goo.gl/UEDh1>.

## Retrieving Application Resources

You can retrieve application resources using the `getResources()` method of the application `Context`. The most straightforward way to retrieve a resource is by using its resource identifier, a unique number automatically generated within the `R.java` class. The following example retrieves a `String` instance from the application resources by its resource ID:

```
String greeting = getResources().getString(R.string.hello);
```

We talk more about different types of application resources in Chapter 7, “Managing Application Resources.”

## Accessing Application Preferences

You can retrieve shared application preferences using the `getSharedPreferences()` method of the application `Context`. The `SharedPreferences` class can be used to save simple application data, such as configuration settings or persistent application state information. We talk more about application preferences in Chapter 12, “Using Android Preferences.”

## Accessing Application Files and Directories

You can use the application `Context` to access create and manage application files and directories private to the application as well as those on external storage. We talk more about application file management in Chapter 13, “Working with Files and Directories.”

## Retrieving Application Assets

You can retrieve application resources using the `getAssets()` method of the application `Context`. This returns an `AssetManager` (`android.content.res.AssetManager`) instance that can then be used to open a specific asset by its name.

## Performing Application Tasks with Activities

The Android Activity class (`android.app.Activity`) is core to any Android application. Much of the time, you define and implement an `Activity` class for each screen in your application. For example, a simple game application might have the following five activities, as shown in Figure 5.1:

- **A startup or splash screen:** This activity serves as the primary entry point to the application. It displays the application name and version information and transitions to the main menu after a short interval.
- **A main menu screen:** This activity acts as a switch to drive the user to the core activities of the application. Here the users must choose what they want to do within the application.
- **A game play screen:** This activity is where the core game play occurs.
- **A high scores screen:** This activity might display game scores or settings.
- **A help/about screen:** This activity might display the information the user might need to play the game.

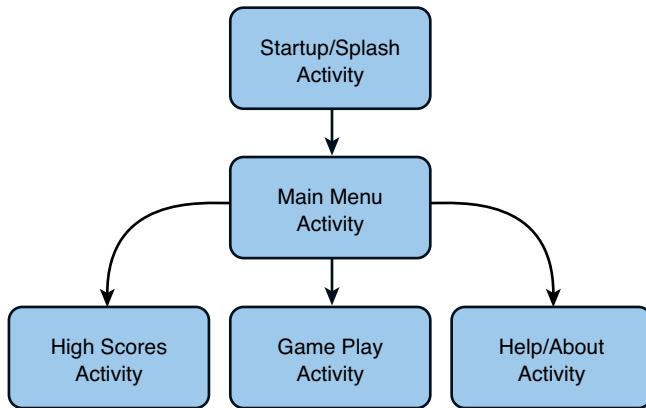


Figure 5.1 A simple game with five activities.

## The Lifecycle of an Android Activity

Android applications can be multiprocess, and the Android operating system allows multiple applications to run concurrently, provided memory and processing power is available. Applications can have background behavior, and applications can be interrupted and paused when events such as phone calls occur. There can be only one active application

visible to the user at a time—specifically, a single application **Activity** is in the foreground at any given time.

The Android operating system keeps track of all **Activity** objects running by placing them on an **Activity** stack (see Figure 5.2). When a new **Activity** starts, the **Activity** on the top of the stack (the current foreground **Activity**) pauses, and the new **Activity** pushes onto the top of the stack. When that **Activity** finishes, it is removed from the **Activity** stack, and the previous **Activity** in the stack resumes.

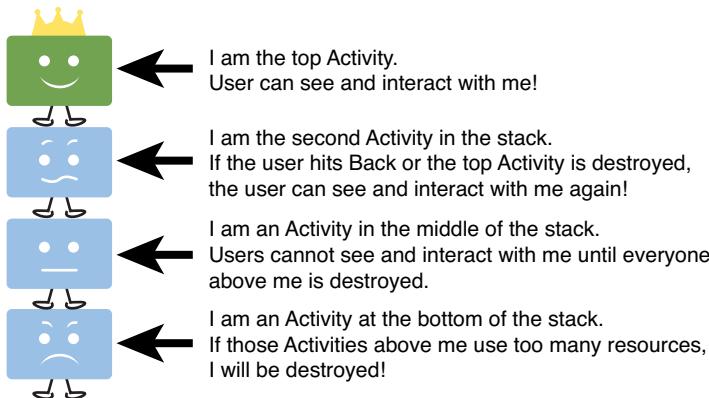


Figure 5.2 The Activity stack.

Android applications are responsible for managing their state and their memory, resources, and data. They must pause and resume seamlessly. Understanding the different states within the **Activity** lifecycle is the first step in designing and developing robust Android applications.

### Using Activity Callbacks to Manage Application State and Resources

Different important state changes within the **Activity** lifecycle are punctuated by a series of important method callbacks. These callbacks are shown in Figure 5.3.

Here are the method stubs for the most important callbacks of the **Activity** class:

```
public class MyActivity extends Activity {  
    protected void onCreate(Bundle savedInstanceState);  
    protected void onStart();  
    protected void onRestart();  
    protected void onResume();  
    protected void onPause();  
    protected void onStop();  
    protected void onDestroy();  
}
```

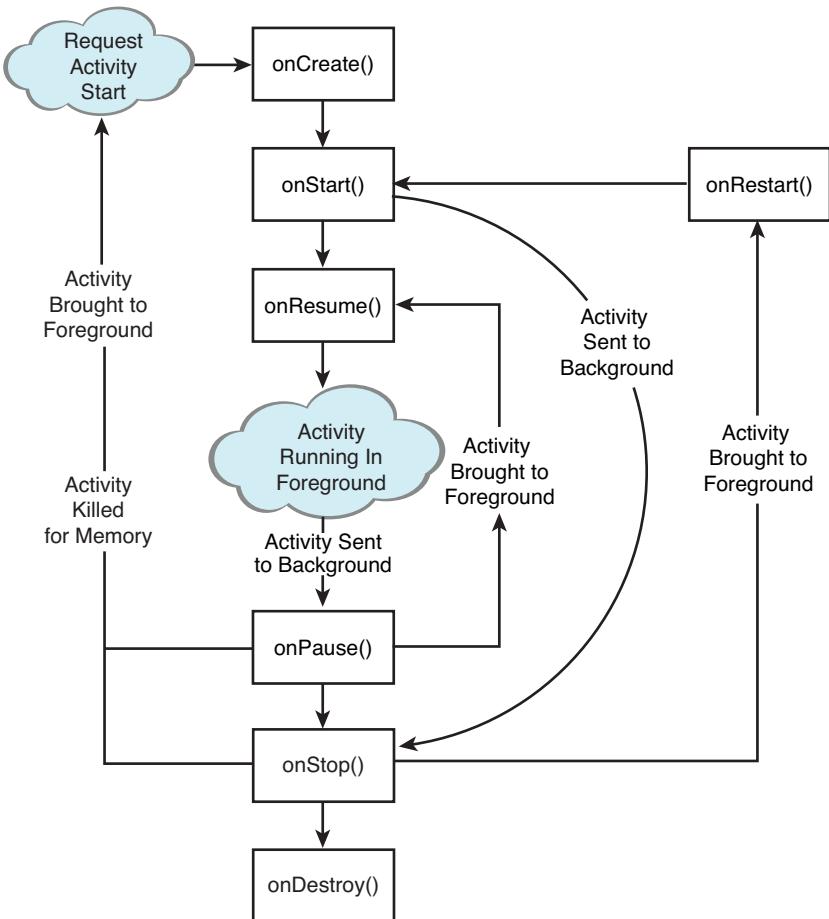


Figure 5.3 The lifecycle of an Android Activity.

Now let's look at each of these callback methods, when they are called, and what they are used for.

### Initializing Static Activity Data in `onCreate()`

When an Activity first starts, the `onCreate()` method is called. The `onCreate()` method has a single parameter, a `Bundle`, which is `null` if this is a newly started Activity. If this Activity was killed for memory reasons and is now restarted, the `Bundle` contains the previous state information for this Activity so that it can reinitialize. It is appropriate to perform any setup, such as layout and data binding, in the `onCreate()` method. This includes calls to the `setContentView()` method.

### Initializing and Retrieving Activity Data in `onResume()`

When the `Activity` reaches the top of the activity stack and becomes the foreground process, the `onResume()` method is called. Although the `Activity` might not be visible yet to the user, this is the most appropriate place to retrieve any instances to resources (exclusive or otherwise) that the `Activity` needs to run. Often, these resources are the most process-intensive, so we only keep these around while the `Activity` is in the foreground.

#### Tip

The `onResume()` method is often the appropriate place to start audio, video, and animations.

### Stopping, Saving, and Releasing Activity Data in `onPause()`

When another `Activity` moves to the top of the activity stack, the current `Activity` is informed that it is being pushed down the activity stack by way of the `onPause()` method.

Here, the `Activity` should stop any audio, video, and animations it started in the `onResume()` method. This is also where you must deactivate resources such as database cursor objects or other objects that should be cleaned up should your activity be terminated. The `onPause()` method may be the last chance for the `Activity` to clean up and release any resources it does not need while in the background. You need to save any uncommitted data here, in case your application does not resume. The system reserves the right to kill an activity without further notice after the call on `onPause()`.

The `Activity` can also save state information to activity-specific preferences or application-wide preferences. We talk more about preferences in Chapter 12.

The `Activity` needs to perform anything in the `onPause()` method in a timely fashion, because the new foreground `Activity` is not started until the `onPause()` method returns.

#### Warning

Generally speaking, any resources and data retrieved in the `onResume()` method should be released in the `onPause()` method. If they aren't, there is a chance that these resources can't be cleanly released if the process is terminated.

### Avoiding Activities Being Killed

Under low-memory conditions, the Android operating system can kill the process for any `Activity` that has been paused, stopped, or destroyed. This essentially means that any `Activity` not in the foreground is subject to a possible shutdown.

If the `Activity` is killed after `onPause()`, the `onStop()` and `onDestroy()` methods will not be called. The more resources released by an `Activity` in the `onPause()`

method, the less likely the Activity is to be killed while in the background without further state methods being called.

The act of killing an Activity does not remove it from the activity stack. Instead, the Activity state is saved into a Bundle object, assuming the Activity implements and uses `onSaveInstanceState()` for custom data, although some View data is automatically saved. When the user returns to the Activity later, the `onCreate()` method is called again, this time with a valid Bundle object as the parameter.



### Tip

So why does it matter if your application is killed when it is straightforward to resume?

Well, it's primarily about responsiveness. The application designer must strike a delicate balance between maintaining data and the resources the application needs to resume quickly, without degrading the CPU and system resources while paused in the background.

### Saving Activity State into a Bundle with `onSaveInstanceState()`

If an Activity is vulnerable to being killed by the Android operating system due to low memory, the Activity can save state information to a Bundle object using the `onSaveInstanceState()` callback method. This call is not guaranteed under all circumstances, so use the `onPause()` method for essential data commits. What we recommend doing is saving important data to persistent storage in `onPause()`, but using `onSaveInstanceState()` to start any data that can be used to rapidly restore the current screen to the state it is in (as the name of the method might imply).



### Tip

You might want to use the `onSaveInstanceState()` method to store nonessential information such as uncommitted form field data or any other state information that might make the user's experience with your application less cumbersome.

When this Activity is returned to later, this Bundle is passed into the `onCreate()` method, allowing the Activity to return to the exact state it was in when the Activity paused. You can also read Bundle information after the `onStart()` callback method using the `onRestoreInstanceState()` callback. Thus, when the Bundle information is there, restoring the previous state will be faster and more efficient than from scratch.

### Destroying Static Activity Data in `onDestroy()`

When an Activity is being destroyed in the normal course of operation, the `onDestroy()` method is called. The `onDestroy()` method is called for one of two reasons: The Activity has completed its lifecycle voluntarily, or the Activity is being killed by the Android operating system because it needs the resources, but still has the time to gracefully destroy your Activity (as opposed to terminating it without calling the `onDestroy()` method).

 Tip

The `isFinishing()` method returns `false` if the Activity has been killed by the Android operating system. This method can also be helpful in the `onPause()` method to know if the Activity is not going to resume right away. However, the Activity might still be killed in the `onStop()` method at a later time, regardless. You may be able to use this as a hint as to how much instance state information to save or permanently persist.

## Organizing Activity Components with Fragments

Until recent versions of the Android SDK, usually a one-to-one relationship existed between an Activity class and an application screen. In other words, for each screen in your app, you defined an Activity to manage its user interface. This worked well enough for small-screened devices such as smartphones, but when the Android SDK started adding support for other types of devices such as tablets and televisions, this relationship did not prove flexible enough. There were times when screen functionality needed to be componentized at a lower level than the Activity class.

Therefore, in Android 3.0, a new concept called fragments was introduced. A **fragment** is a chunk of screen functionality or user interface with its own lifecycle that can exist within an activity and is represented by the `Fragment` class (`android.app.Fragment`) and several supporting classes. A `Fragment` class instance must exist within an Activity instance (and its lifecycle), but fragments need not be paired with the same Activity class each time it's instantiated.

 Tip

You may be wondering why we consider fragments a fundamental building block of Android applications if they were only recently included in the Android 3.0 SDK. Everything we've heard and read from the developers of the Android SDK implies that fragments are the future of Android application design across all device types. Certain more established user interface components have been deprecated, with new versions introduced within the Fragment APIs. The Android SDK documentation heavily promotes their use, to the extent that the Android SDK now includes a Compatibility package (also called the Support Package) that enables fragment library usage on all currently used Android platform versions (as far back as Android 1.6).

Fragments, and how they make applications more flexible, are best illustrated by example. Consider a simple MP3 music player application that allows the user to view a list of artists, drill down to list their albums, and drill down further to see each track in an album. When the user chooses to play a song at any point, that track's album art is displayed along with the track info and progress (with next, previous, pause, and so on).

Now, if you were using the simple one-screen-to-one-activity rule of thumb, you'd count four screens here, which could be called: List Artists, List Artist Albums, List Album Tracks, and Show Track. You could implement four activities, one for each screen. This

would likely work just fine for a small-screen device such as a smartphone. But on a tablet or a television, you’re wasting a whole lot of space. Or, thought another way, you have the opportunity to provide a much richer user experience on a device with more screen real estate. Indeed, on a large enough screen, you might want to implement a standard music library interface:

- Column #1 displays a list of artists. Selecting an artist filters the second column.
- Column #2 displays a list of that artist’s albums. Selecting an album filters the third column.
- Column #3 displays a list of that album’s tracks.
- The bottom half the screen, below the columns, always displays the artist, album, or track art and details, depending on what’s selected in the columns above. If the user ever chooses the “Play” function, the application can display the track info and progress in this area of the screen as well.

This sort of application design only requires a single screen, and thus a single `Activity` class, as shown in Figure 5.4.

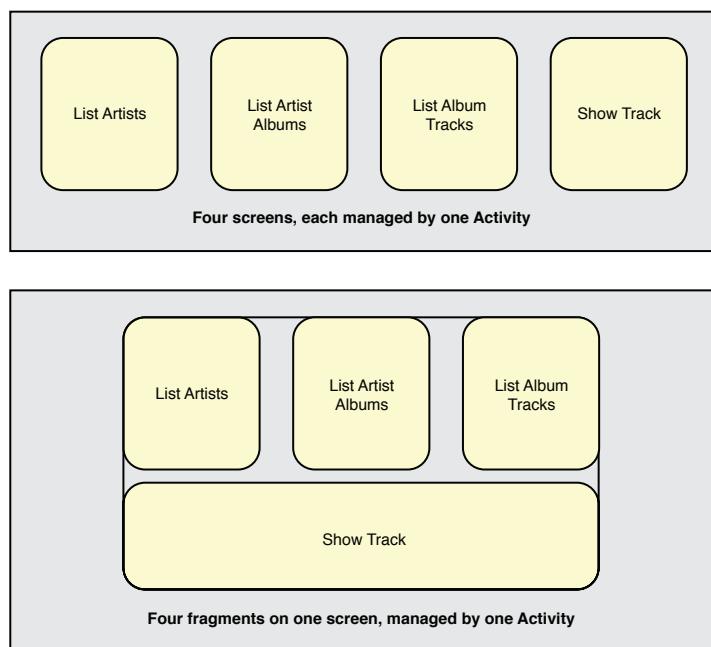


Figure 5.4 How fragments can improve application workflow flexibility.

But then you’re stuck with having to develop basically two separate applications: one to work on smaller screens, and another to work on larger ones. This is where fragments

come in. If you componentize your features and make four fragments (list artists, list albums, list track, show play track), you can mix and match them on the fly, while still having only one codebase to maintain.

We discuss fragments in detail in Chapter 10, “Working with Fragments.”

## Managing Activity Transitions with Intents

In the course of the lifetime of an Android application, the user might transition between a number of different `Activity` instances. At times, there might be multiple `Activity` instances on the activity stack. Developers need to pay attention to the life-cycle of each `Activity` during these transitions.

Some `Activity` instances—such as the application splash/startup screen—are shown and then permanently discarded when the main menu screen `Activity` takes over. The user cannot return to the splash screen `Activity` without re-launching the application. In this case, use the `startActivity()` and appropriate `finish()` methods.

Other `Activity` transitions are temporary, such as a child `Activity` displaying a dialog and then returning to the original `Activity` (which was paused on the activity stack and now resumes). In this case, the parent `Activity` launches the child `Activity` and expects a result. For this, use the `startActivityForResult()` and `onActivityResult()` methods.

## Transitioning Between Activities with Intents

Android applications can have multiple entry points. A specific `Activity` can be designated as the main `Activity` to launch by default within the `AndroidManifest.xml` file; we talk more about this file in Chapter 6, “Defining Your Application Using the Android Manifest File.”

Other activities might be designated to launch under specific circumstances. For example, a music application might designate a generic `Activity` to launch by default from the Application menu, but also define specific alternative entry-point activities for accessing specific music playlists by playlist ID or artists by name.

### Launching a New Activity by Class Name

You can start activities in several ways. The simplest method is to use the `Application Context` object to call the `startActivity()` method, which takes a single parameter, an `Intent`.

An `Intent` (`android.content.Intent`) is an asynchronous message mechanism used by the Android operating system to match task requests with the appropriate `Activity` or `Service` (launching it, if necessary) and to dispatch broadcast `Intent` events to the system at large.

For now, though, we focus on the `Intent` object and how it is used with activities. The following line of code calls the `startActivity()` method with an explicit

`Intent`. This `Intent` requests the launch of the target `Activity` named `MyDrawActivity` by its class. This class is implemented elsewhere within the package.

```
startActivity(new Intent(getApplicationContext(),
    MyDrawActivity.class));
```

This line of code might be sufficient for some applications, which simply transition from one `Activity` to the next. However, you can use the `Intent` mechanism in a much more robust manner. For example, you can use the `Intent` structure to pass data between activities.

### **Creating Intents with Action and Data**

You've seen the simplest case to use an `Intent` to launch a class by name. `Intents` need not specify the component or class they want to launch explicitly. Instead, you can create an `Intent Filter` and register it within the `Android Manifest` file. An `Intent Filter` is used by activities, services, and broadcast receivers to specify which intents each is interested in receiving (and filtering out the rest). The `Android` operating system attempts to resolve the `Intent` requirements and launch the appropriate `Activity` based on the filter criteria.

The guts of the `Intent` object are composed of two main parts: the *action* to be performed and, optionally, the *data* to be acted upon. You can also specify action/data pairs using `Intent` `Action` types and `Uri` objects. As you saw in Chapter 3, "Writing Your First `Android` Application," a `Uri` object represents a string that gives the location and name of an object. Therefore, an `Intent` is basically saying "do this" (the *action*) to "that" (the `Uri` describing what resource to do the *action* to).

The most common action types are defined in the `Intent` class, including `ACTION_MAIN` (describes the main entry point of an `Activity`) and `ACTION_EDIT` (used in conjunction with a `Uri` to the data edited). You also find `Action` types that generate integration points with activities in other applications, such as the `Browser` or `Phone Dialer`.

### **Launching an Activity Belonging to Another Application**

Initially, your application might be starting only activities defined within its own package. However, with the appropriate permissions, applications might also launch external activities within other applications. For example, a Customer Relationship Management (CRM) application might launch the `Contacts` application to browse the Contact database, choose a specific contact, and return that contact's unique identifier to the CRM application for use.

Here is an example of how to create a simple `Intent` with a predefined `Action` (`ACTION_DIAL`) to launch the `Phone Dialer` with a specific phone number to dial in the form of a simple `Uri` object:

```
Uri number = Uri.parse("tel:5555551212");
Intent dial = new Intent(Intent.ACTION_DIAL, number);
startActivity(dial);
```

You can find a list of commonly used Google application intents at <http://d.android.com/guide/appendix/g-app-intents.html>. Also available is the developer-managed Registry of Intents protocols at OpenIntents, found at <http://www.openintents.org/en/intentstable>. A growing list of intents is available from third-party applications and those within the Android SDK.

## Passing Additional Information Using Intents

You can also include additional data in an Intent. The Extras property of an Intent is stored in a Bundle object. The Intent class also has a number of helper methods for getting and setting name/value pairs for many common data types.

For example, the following Intent includes two extra pieces of information—a string value and a boolean:

```
Intent intent = new Intent(this, MyActivity.class);
intent.putExtra("SomeStringData", "Foo");
intent.putExtra("SomeBooleanData", false);
startActivity(intent);
```

Then in the onCreate() method of the MyActivity class, you can retrieve the extra data sent as follows:

```
Bundle extras = getIntent().getExtras();
if (extras != null) {
    String myStr = extras.getString("SomeStringData", "Default Value");
    Boolean myBool = extras.getBoolean("SomeBooleanData", true);
}
```

### Tip

The strings you use to identify your Intent object extras can be whatever you want. However, the Android convention for the key name for “extra” data is to include a package prefix—for example, com.androidbook.Multimedia.SomeStringData. We also recommend defining the extra string names in the Activity for which they are used. (We’ve skipped doing this in the preceding example to keep it short.)

## Organizing Application Navigation with Activities and Intents

As previously mentioned, your application likely has a number of screens, each with its own Activity. There is a close relationship between activities and intents, and application navigation. You often see a kind of menu paradigm used in several different ways for application navigation:

- **Main menu or list-style screen:** Acts as a switch in which each menu item launches a different Activity in your application. For instance, menu items for launching the Play Game Activity, the High Scores Activity, and the Help Activity.

- **Drilldown list-style screen:** Acts as a directory in which each menu item launches the same Activity, but each item passes in different data as part of the Intent (for example, a menu of all database records). Choosing a specific item might launch the Edit Record Activity, passing in that particular item's unique identifier.
- **Click actions:** Sometimes you want to navigate between screens in the form of a wizard. You might set the click handler for a user interface control, such as a "Next" button, to trigger a new activity to start and the current one to finish.
- **Options menus:** Some applications like to hide their navigational options until the user needs them. The user can then click the Menu button on the device and launch an options menu, where each option listed corresponds to an intent to launch a different activity.
- **Action bar-style navigation:** Introduced in a recent version of the Android SDK, action bars are a functional title bar with navigational button options, each of which spawn an intent and launch a specific activity. We talk more about action bars in *Android Wireless Application Development Volume II: Advanced Topics*.

## Working with Services

Trying to wrap your head around activities and intents when you start with Android development can be daunting. We have tried to distill everything you need to know to start writing Android applications with multiple Activity classes, but we'd be remiss if we didn't mention that there's a lot more here, much of which is discussed throughout the two volumes of this book using practical examples. However, we need to give you a "heads up" about some of these topics now because we begin to touch on them in the next chapter when we cover configuring the Android Manifest file for your application.

One application component we have briefly discussed is the service. An Android Service (`android.app.Service`) can be thought of as a developer-created component that has no user interface of its own. An Android Service can be one of two things, or both. It can be used to perform lengthy operations that may go beyond the scope of a single activity. Additionally, a service can be the server of a client/server for providing functionality through remote invocation via interprocess communication. Although often used to control long-running server operations, the processing could be whatever the developer wants. Any Service classes exposed by an Android application must be registered in the Android Manifest file.

You can use services for different purposes. Generally, you use a service when no input is required from the user. Here are some circumstances in which you might want to implement or use an Android service:

- A weather, email, or social network app might implement a service to routinely check for updates on the network. (Note: There are other implementations for polling, but this is a common use of services.)

- A game might create a service to download and process the content for the next level in advance of when the user needs it.
- A photo or media app that keeps its data in sync online might implement a service to package and upload new content in the background when the device is idle.
- A video-editing app might offload heavy processing to a queue on its service in order to avoid affecting overall system performance for nonessential tasks.
- A news application might implement a service to “pre-load” content by downloading news stories in advance of when the user launches the application, to improve performance and responsiveness.

A good rule of thumb is that if the task requires the use of a worker thread, might affect application responsiveness and performance, and is not time sensitive to the application, consider implementing a service to handle the task outside the main application and any individual activity lifecycles. We have a whole chapter devoted to services in *Android Wireless Application Development Volume II: Advanced Topics*.

## Receiving and Broadcasting Intents

Intents serve yet another purpose. You can broadcast an `Intent` (via a call to `broadcastIntent()`) to the Android system at large, allowing any interested application (called a `BroadcastReceiver`) to receive that broadcast and act upon it. Your application might send off as well as listen for `Intent` broadcasts. Broadcasts are generally used to inform the system that something interesting has happened. For example, a commonly listened-for broadcast intent is `ACTION_BATTERY_LOW`, which is broadcast as a warning when the battery is low. If your application has a battery-hogging service of some kind, or might lose data in the event of an abrupt shutdown, it might want to listen for this kind of broadcast and act accordingly. There are also broadcast events for other interesting system events, such as SD card state changes, applications being installed or removed, and the wallpaper being changed.

Your application can also share information using this same broadcast mechanism. For example, an email application might broadcast an intent whenever a new email arrives so that other applications (such as spam or antivirus apps) that might be interested in this type of event can react to it.

We talk more about broadcasting, working with hardware APIs, and the battery in *Android Wireless Application Development Volume II: Advanced Topics*.

## Summary

We have tried to strike a balance between providing a thorough reference without overwhelming you with details you won’t need to know when developing a typical Android application. Instead, we have focused on the details you need to know to move forward developing Android applications and to understand every example provided within this book.

The `Activity` class is the core building block of any Android application. Each `Activity` performs a specific task within the application, generally represented by a single screen. Each `Activity` is responsible for managing its own resources and data through a series of lifecycle callbacks. Meanwhile, you can break your `Activity` class into functional components using the `Fragment` class. This allows more than one `Activity` to display similar components of a screen without duplicating code across multiple `Activity` classes. The transition from one `Activity` to the next is achieved through the `Intent` mechanism. An `Intent` object acts as an asynchronous message that the Android operating system processes and responds to by launching the appropriate `Activity` or `Service`. You can also use `Intent` objects to broadcast systemwide events to any interested applications listening.

## References and More Information

Android SDK Reference regarding the application `Context` class:

<http://developer.android.com/reference/android/content/Context.html>

Android SDK Reference regarding the `Activity` class:

<http://developer.android.com/reference/android/app/Activity.html>

Android SDK Reference regarding the `Fragment` class:

<http://developer.android.com/reference/android/app/Fragment.html>

Android Dev Guide: "Fragments":

<http://developer.android.com/guide/topics/fundamentals/fragments.html>

Using the Android Compatibility (Support) Package:

<http://d.android.com/sdk/compatibility-library.html>

Android Dev Guide: "Intents and Intent Filters":

<http://developer.android.com/guide/topics/intents/intents-filters.html>

# 6

## Defining Your Application Using the Android Manifest File

Android projects use a special configuration file called the Android manifest file to determine application settings—settings such as the application name and version, as well as what permissions the application requires to run and what application components it is composed of. In this chapter, you explore the Android manifest file in detail and learn how applications use it to define and describe application behavior.

### Configuring Android Applications Using the Android Manifest File

The Android application manifest file is a specially formatted XML file that must accompany each Android application. This file contains important information about the application’s identity. Here you define the application’s name and version information as well as what application components the application relies upon, what permissions the application requires to run, and other application configuration information.

The Android manifest file is named `AndroidManifest.xml` and must be included at the top level of any Android project. The information in this file is used by the Android system to

- Install and upgrade the application package.
- Display the application details, such as the application name, description, and icon to users.
- Specify application system requirements, including which Android SDKs are supported, what device configurations are required (for example, d-pad navigation), and which platform features the application relies upon (for example, uses multi-touch capabilities).
- Specify what features are required by the application for market-filtering purposes.

- Register application activities and when they should be launched.
- Manage application permissions.
- Configure other advanced application component configuration details, including defining services, broadcast receivers, and content providers.
- Specify intent filters for your activities, services, and broadcast receivers.
- Enable application settings such as debugging and configuring instrumentation for application testing.



### Tip

When you use Eclipse with the Android Plug-In for Eclipse (ADT), the Android Project Wizard creates the initial `AndroidManifest.xml` file for you. If you are not using Eclipse, the `android` command-line tool creates the Android manifest file for you as well.

## Editing the Android Manifest File

The manifest resides at the top level of your Android project. You can edit the Android manifest file using the Eclipse Manifest File resource editor, which is a feature of the Android ADT plug-in for Eclipse, or by manually editing the XML.



### Tip

For simple configuration changes, we recommend using the editor. However, if we are adding a bunch of activity registrations, or something more complex, we usually edit the XML directly because the resource editor can be somewhat confusing and no support documentation is available. We've found that when a more complex configuration results in nested XML (such as an intent filter, for example), then it's too easy for users to end up with manifest settings at the wrong level in the XML tag hierarchy. Therefore, if you use the editor, you should always spot-check the resulting XML to make sure it looks correct.

## Editing the Manifest File Using Eclipse

You can use the Eclipse Manifest File resource editor to edit the project manifest file. The Eclipse Manifest File resource editor organizes the manifest information into categories:

- The Manifest tab
- The Application tab
- The Permissions tab
- The Instrumentation tab
- The `AndroidManifest.xml` tab

Let's take a closer look at a sample Android manifest file. We've chosen a more complex sample project from the second volume of this book because it illustrates a number of different characteristics of the Android manifest file, as opposed to the very simple default manifest file you configured for the MyFirstAndroidApp project.

### Configuring Package-Wide Settings Using the Manifest Tab

The Manifest tab (see Figure 6.1) contains package-wide settings, including the package name, version information, and supported Android SDK information. You can also set any hardware or feature requirements here.

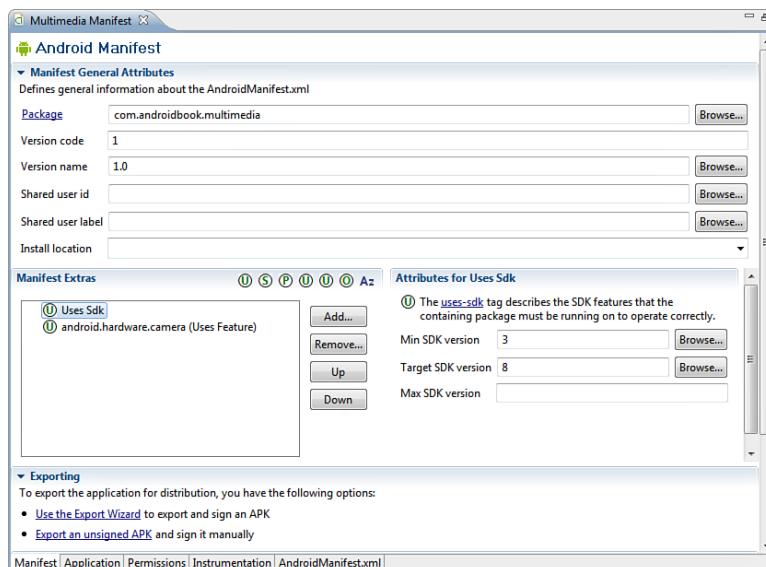


Figure 6.1 The Manifest tab of the Eclipse Manifest File resource editor.

### Managing Application and Activity Settings Using the Application Tab

The Application tab (see Figure 6.2) contains application-wide settings, including the application label and icon, as well as information about the application components, such as activities, and other application components, including configuration for services, intent filters, and content providers.

### Enforcing Application Permissions Using the Permissions Tab

The Permissions tab (see Figure 6.3) contains any permission rules required by your application. This tab can also be used to enforce custom permissions created for the application.

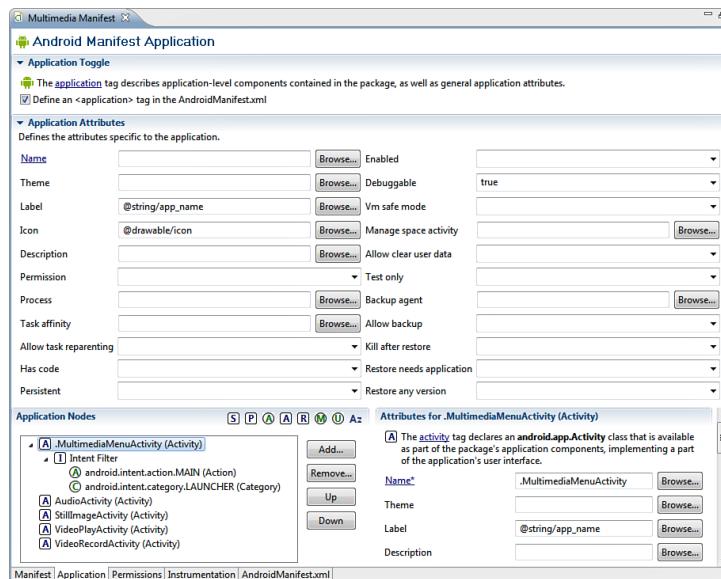


Figure 6.2 The Application tab of the Eclipse Manifest File resource editor.

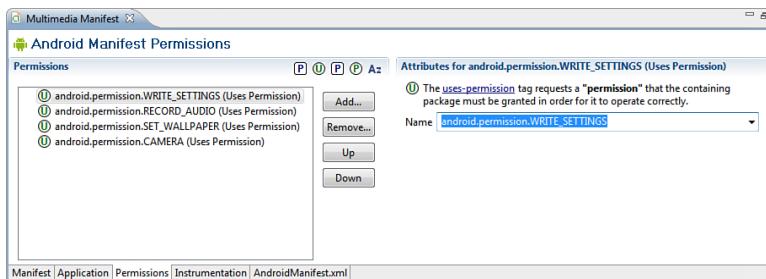


Figure 6.3 The Permissions tab of the Eclipse Manifest File resource editor.



### Warning

Do not confuse the application Permission field (a drop-down list on the Application tab) with the Permissions tab features. Use the Permissions tab to define the permissions required by the application.

### Managing Test Instrumentation Using the Instrumentation Tab

The Instrumentation tab allows the developer to declare any instrumentation classes for monitoring the application. We talk more about instrumentation and testing in Chapter 18, “Testing Android Applications.”

## Editing the Manifest File Manually

The Android manifest file is a specially formatted XML file. You can edit the XML manually by clicking the `AndroidManifest.xml` tab.

Android manifest files generally include a single `<manifest>` tag with a single `<application>` tag. The following is a sample `AndroidManifest.xml` file for an application called Multimedia:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.multimedia"
    android:versionCode="1"
    android:versionName="1.0">
    <application android:icon="@drawable/icon"
        android:label="@string/app_name"
        android:debuggable="true">
        <activity android:name=".MultimediaMenuActivity"
            android:label="@string/app_name">
            <intent-filter>
                <action
                    android:name="android.intent.action.MAIN" />
                <category
                    android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name="AudioActivity"></activity>
        <activity android:name="StillImageActivity"></activity>
        <activity android:name="VideoPlayActivity"></activity>
        <activity android:name="VideoRecordActivity"></activity>
    </application>
    <uses-permission
        android:name="android.permission.WRITE_SETTINGS" />
    <uses-permission
        android:name="android.permission.RECORD_AUDIO" />
    <uses-permission
        android:name="android.permission.SET_WALLPAPER" />
    <uses-permission
        android:name="android.permission.CAMERA"></uses-permission>
    <uses-sdk
        android:minSdkVersion="4"
        android:targetSdkVersion="10">
    </uses-sdk>
    <uses-feature
        android:name="android.hardware.camera" />
</manifest>
```

Here's a summary of what this file tells us about the Multimedia application:

- The application uses the package name `com.androidbook.multimedia`.
- The application version name is 1.0.
- The application version code is 1.
- The application name and label are stored in the resource string called `@string/app_name` within the `/res/values/strings.xml` resource file.
- The application is debuggable on an Android device.
- The application icon is the graphic file called `icon` (could be a PNG, JPG, or GIF) stored within the `/res/drawable` directory (there are actually multiple versions for different pixel densities).
- The application has five activities (`MultimediaMenuActivity`, `AudioActivity`, `StillImageActivity`, `VideoPlayActivity`, and `VideoRecordActivity`).
- `MultimediaMenuActivity` is the primary entry point for the application because it handles the action `android.intent.action.MAIN`. This activity shows in the application launcher, because its category is `android.intent.category.LAUNCHER`.
- The application requires the following permissions to run: the ability to record audio, the ability to set the wallpaper on the device, the ability to access the built-in camera, and the ability to write settings.
- The application works from any API level from 4 to 10; in other words, Android SDK 1.6 is the lowest supported platform version, and the application was written to target Gingerbread MR1 (for example, Android 2.3.4).
- Finally, the application requires a camera to work properly.

Now let's talk about some of these important configurations in detail.

## Managing Your Application's Identity

Your application's Android manifest file defines the application properties. The package name must be defined in the Android manifest file within the `<manifest>` tag using the `package` attribute:

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

## Versioning Your Application

Versioning your application appropriately is vital to maintaining your application in the field. Intelligent versioning can help reduce confusion and make product support and upgrades simpler. Two different version attributes are defined within the `<manifest>` tag: the version name and the version code.

The version name (`android:versionName`) is a user-friendly, developer-defined version attribute. This information is displayed to users when they manage applications on their devices and when they download the application from marketplaces. Developers use this version information to keep track of their application versions in the field. We discuss appropriate application versioning for mobile applications in detail in Chapter 16, “The Android Software Development Process.”

### Warning

Although you can use an `@string` resource reference for some manifest file settings, such as the `android:versionName`, some publishing systems don’t support this.

The Android operating system uses the version code (`android:versionCode`), which is a numeric attribute, to manage application upgrades. We talk more about publishing and upgrade support in Chapter 19, “Publishing Your Android Application.”

## Setting the Application Name and Icon

Overall application settings are configured with the `<application>` tag of the Android manifest file. Here you set information such as the application icon (`android:icon`) and friendly name (`android:label`). These settings are attributes of the `<application>` tag.

For example, here we set the application icon to an image resource provided with the application package and the application label to a string resource:

```
<application android:icon="@drawable/icon"  
    android:label="@string/app_name">
```

You can also set optional application settings as attributes in the `<application>` tag, such as the application description (`android:description`) and the setting to enable the application for debugging on the device (`android:debuggable="true"`).

## Enforcing Application System Requirements

In addition to configuring your application’s identity, the Android manifest file is also used to specify any system requirements necessary for the application to run properly. For example, an augmented reality application might require that the device have GPS, a compass, and a camera. Similarly, an application that relies on the Bluetooth APIs available within the Android SDK requires a device with an SDK version of API Level 5 or higher (Android 2.0), because that’s where those APIs were introduced.

These types of system requirements can be defined and enforced in the Android manifest file. When an application is installed on a device, the Android platform checks these requirements and will error out if necessary. Similarly, the Android Market uses information in the Android manifest file to filter which applications to offer to which devices so that users are installing applications that should work on their devices.

Some of the application system requirements that developers can configure through the Android manifest file include

- The Android SDK versions supported by the application
- The Android platform features used by the application
- The Android hardware configurations required by the application
- The screen sizes and pixel densities supported by the application
- Any external libraries that the application links to

## Targeting Specific SDK Versions

Android devices run different versions of the Android platform. Often, you see old, less powerful, or even less expensive devices running older versions of the Android platform, whereas newer, more powerful devices that show up on the market often run the latest Android software.

There are now hundreds of different Android devices in users' hands. Developers must decide who their target audience is for a given application. Are they trying to support the largest population of users and therefore want to support as many different versions of the platform as possible? Or are they developing a bleeding-edge game that requires the latest device hardware?



### Tip

You can expand the range of target SDKs that your application supports by using the compatibility package, or by using Java reflection to check for SDK features before using them. You can read more about the compatibility package at <http://goo.gl/WcltO> and more about backward compatibility at <http://goo.gl/oTNUJ> and <http://goo.gl/L6iZR>.

Developers can specify which versions of the Android platform an application supports within its Android manifest file using the `<uses-sdk>` tag. This tag has three important attributes:

- **The `minSdkVersion` attribute:** This attribute specifies the lowest API level that the application supports.
- **The `targetSdkVersion` attribute:** This attribute specifies the optimum API level that the application supports.
- **The `maxSdkVersion` attribute:** This attribute specifies the highest API level that the application supports.



### Tip

The Android Market filters applications available to a given user based on settings such as the `<uses-sdk>` tag within an application's manifest file. This is a required tag for applications that want to be published on the Android Market. Neglecting to use this tag results in a warning in the build environment.

Each attribute of the `<uses-sdk>` tag is an integer that represents the API level associated with a given Android SDK. This value does not directly correspond to the SDK version. Instead, it is the revision of the API level associated with that SDK. The API level is set by the developers of the Android SDK. You need to check the SDK documentation to determine the API level value for each version. Table 6.1 shows the Android SDK versions available for shipping applications.

Table 6.1 Android SDK Versions and Their API Levels

Android SDK Version	API Level	Codename / Version Code
Android 1.0	1	BASE
Android 1.1	2	BASE_1_1
Android 1.5	3	CUPCAKE
Android 1.6	4	DONUT
Android 2.0	5	ÉCLAIR
Android 2.0.1	6	ECLAIR_0_1
Android 2.1.X	7	ECLAIR_MR1
Android 2.2.X	8	FROYO
Android 2.3, 2.3.1, 2.3.2	9	GINGERBREAD
Android 2.3.3, 2.3.4	10	GINGERBREAD_MR1
Android 3.0.X	11	HONEYCOMB
Android 3.1.X	12	HONEYCOMB_MR1
Android 3.2	13	HONEYCOMB_MR2
Android 4.0, 4.0.1, 4.0.2	14	ICE_CREAM SANDWICH
Android 4.0.3	15	ICE_CREAM SANDWICH_MR1

### Specifying the Minimum SDK Version

You should always specify the `minSdkVersion` attribute for your application. This value represents the lowest Android SDK version your application supports.

For example, if your application requires APIs introduced in Android SDK 1.6, you would check that SDK's documentation and find that this release is defined as API Level 4. Therefore, add the following to your Android manifest file within the `<manifest>` tag block:

```
<uses-sdk android:minSdkVersion="4" />
```

It's that simple. You should use the lowest API level possible if you want your application to be compatible with the largest number of Android devices. However, you must ensure that your application is tested sufficiently on any non-target platforms (any API level supported below your target SDK, as described in the next section).

### Specifying the Target SDK Version

You should always specify the `targetSdkVersion` attribute for your application. This value represents the Android SDK version your application was built for and tested against.

For example, if your application was built using APIs that are backward-compatible to Android 1.6 (API Level 4), but targeted and tested using Android 2.3.4 SDK (API Level 10), then you would want to specify the `targetSdkVersion` attribute as 10. Therefore, add the following to your Android manifest file within the `<manifest>` tag block:

```
<uses-sdk android:minSdkVersion="4" android:targetSdkVersion="10" />
```

Why should you specify the target SDK version you used? Well, the Android platform has built-in functionality for backward compatibility (to a point). Think of it like this: A specific method of a given API might have been around since API Level 1. However, the internals of that method—its behavior—might have changed slightly from SDK to SDK. By specifying the target SDK version for your application, the Android operating system attempts to match your application with the exact version of the SDK (and the behavior as you tested it within the application), even when running a different (newer) version of the platform. This means that the application should continue to behave in “the old way” despite any new changes or “improvements” to the SDK that might cause unintended consequences in your application.

### Specifying the Maximum SDK Version

You will rarely want to specify the `maxSdkVersion` attribute for your application. This value represents the highest Android SDK version your application supports, in terms of API level. It restricts forward compatibility of your application.

One reason you might want to set this attribute is if you want to limit who can install the application to exclude devices with the newest SDKs. For example, you might develop a free beta version of your application with plans for a paid version for the newest SDK. By setting the `maxSdkVersion` attribute of the manifest file for your free application, you disallow anyone with the newest SDK to install the free version of the application. The downside of this idea? If your users have devices that receive over-the-air SDK updates, your application would cease to work (and appear) on devices where it had functioned perfectly, which might “upset” your users and result in bad ratings on your market of choice. In short, use `maxSdkVersion` only when absolutely necessary and when you understand the risks associated with its use.

## Enforcing Application Platform Requirements

Android devices have different hardware and software configurations. Some devices have built-in keyboards and others rely on the software keyboard. Similarly, certain Android devices support the latest 3D graphics libraries and others provide little or no graphics support. The Android manifest file has several informational tags for flagging the system features and hardware configurations supported or required by an Android application.

### Specifying Supported Input Methods

The `<uses-configuration>` tag can be used to specify which hardware and software input methods the application supports. There are different configuration attributes for five-way navigation: the hardware keyboard and keyboard types; navigation devices such as the directional pad, trackball, and wheel; and touch screen settings.

There is no “OR” support within a given attribute. If an application supports multiple input configurations, there must be multiple `<uses-configuration>` tags defined in your Android manifest file—one for each configuration supported.

For example, if your application requires a physical keyboard and touch screen input using a finger or a stylus, you need to define two separate `<uses-configuration>` tags in your manifest file, as follows:

```
<uses-configuration android:reqHardKeyboard="true"
    android:reqTouchScreen="finger" />
<uses-configuration android:reqHardKeyboard="true"
    android:reqTouchScreen="stylus" />
```

For more information about the `<uses-configuration>` tag of the Android manifest file, see the Android SDK reference at <http://d.android.com/guide/topics/manifest/uses-configuration-element.html>.

### Specifying Required Device Features

Not all Android devices support every Android feature. Put another way: There are a number of APIs (and related hardware) that Android device manufacturers and carriers may optionally include. For example, not all Android devices have multitouch capability or a camera flash.

The `<uses-feature>` tag is used to specify which Android features your application uses to run properly. These settings are for informational purposes only—the Android operating system does not enforce these settings, but publication channels such as the Android Market use this information to filter the applications available to a given user. Other applications might check this information as well.

If your application requires multiple features, you must create a `<uses-feature>` tag for each feature. For example, an application that requires both a light and proximity sensor requires two tags:

```
<uses-feature android:name="android.hardware.sensor.light" />
<uses-feature android:name="android.hardware.sensor.proximity" />
```

One common reason to use the `<uses-feature>` tag is for specifying the OpenGL ES versions supported by your application. By default, all applications function with OpenGL ES 1.0 (which is a required feature of all Android devices). However, if your application requires features available only in later versions of OpenGL ES, such as 2.0, then you must specify this feature in the Android manifest file. This is done using the `android:glEsVersion` attribute of the `<uses-feature>` tag. Specify the lowest version of OpenGL ES that the application requires. If the application works with 1.0 and 2.0, specify the lowest version (so that the Android Market allows more users to install your application).

For more information about the `<uses-feature>` tag of the Android manifest file, see the Android SDK reference at <http://d.android.com/guide/topics/manifest/uses-feature-element.html>.

### Specifying Supported Screen Sizes

Android devices come in many shapes and sizes. Screen sizes and pixel densities vary tremendously across the wide range of Android devices available on the market today. The `<supports-screen>` tag can be used to specify which Android types of screens the application supports. The Android platform categorizes screen types in terms of sizes (small, normal, large, and `xlarge`) and pixel density (`LDPI`, `MDPI`, `HDPI`, and `XHDPI`, representing low, medium, high, and extra-high density displays). These characteristics effectively cover the variety of screen types available within the Android platform.

For example, if the application supports QVGA screens (`small`) and HVGA screens (`normal`) regardless of pixel density, the application's `<supports-screen>` tag is configured as follows:

```
<supports-screens android:smallScreens="true"
                  android:normalScreens="true"
                  android:largeScreens="false"
                  android:anyDensity="true" />
```

For more information about the `<supports-screen>` tag of the Android manifest file, see the Android SDK reference at <http://d.android.com/guide/topics/manifest/supports-screens-element.html> as well as the Android Dev Guide on Screen Support at [http://d.android.com/guide/practices/screens\\_support.html#DensityConsiderations](http://d.android.com/guide/practices/screens_support.html#DensityConsiderations).

### Working with External Libraries

You can register any shared libraries your application links to within the Android manifest file. By default, every application is linked to the standard Android packages (such as `android.app`) and is aware of its own package. However, if your application links to additional packages, they should be registered within the `<application>` tag of the Android manifest file using the `<uses-library>` tag. For example:

```
<uses-library android:name="com.sharedlibrary.sharedStuff" />
```

This feature is often used for linking to optional Google APIs. For more information about the `<uses-library>` tag of the Android manifest file, see the Android SDK reference at <http://d.android.com/guide/topics/manifest/uses-library-element.html>.

## Other Application Configuration Settings and Filters

You'll want to be aware of several other lesser used Manifest file settings because they are also used by the Android Market for application filtering:

- The `<supports-gl-texture>` tag is used to specify the GL texture compression format supported by the application. This tag is used by applications that use the graphics libraries and want to be compatible only with devices that support a specific compression format. For more info about this manifest file tag, see the Android SDK documentation at <http://d.android.com/guide/topics/manifest/supports-gl-texture-element.html>.
- The `<compatible-screens>` tag is used solely by the Android Market to restrict installation of your application to devices with specific screen sizes. This tag is not checked by the Android operating system and usage is discouraged unless you absolutely need to restrict the installation of your application on certain devices. For more info about this manifest file tag, see the Android SDK documentation at <http://d.android.com/guide/topics/manifest/compatible-screens-element.html>.

## Registering Activities in the Android Manifest

Each activity within the application must be defined within the Android manifest file with an `<activity>` tag. For example, the following XML excerpt registers an Activity class called `AudioActivity`:

```
<activity android:name="AudioActivity" />
```

This activity must be defined as a class within the `com.androidbook.multimedia` package—that is, the package specified in the `<manifest>` element of the Android manifest file. You can also enforce scope of the Activity class by using the dot as a prefix in the Activity class name:

```
<activity android:name=".AudioActivity" />
```

Or you can specify the complete class name:

```
<activity android:name="com.androidbook.multimedia.AudioActivity" />
```



### Warning

You must define the `<activity>` tag for each Activity or it will not run as part of your application. It is quite common for developers to implement an Activity and then forget to do this. They then spend a lot of time troubleshooting why it isn't running properly, only to realize they forgot to register it in the Android manifest file.

## Designating a Primary Entry Point Activity for Your Application Using an Intent Filter

You can designate an `Activity` class as the primary entry point by configuring an intent filter using the Android manifest tag `<intent-filter>` in the Android manifest file with the `MAIN` action type and the `LAUNCHER` category.

For example, the following XML configures an activity called `MultimediaMainActivity` as the primary launching point of the application:

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

## Configuring Other Intent Filters

The Android operating system uses intent filters to resolve implicit intents—that is, intents that do not specify a specific activity or other component type to launched. Intent filters can be applied to activities, services, and broadcast receivers. An intent filter declares that this application component is capable of handling or processing a specific type of intent when it matches the filter's criteria.

Different applications have the same sorts of intent filters and are able to process the same sorts of requests. In fact, this is how the “share” features and the flexible application launch system of the Android operating system works. For example, you can have several different web browsers installed on a device, all of which can handle “browse the Web” intents by setting up the appropriate filters.

Intent filters are defined using the `<intent-filter>` tag and must contain at least one `<action>` tag but can also contain other information, such as `<category>` and `<data>` blocks. Here we have a sample intent filter block that might be found within an `<activity>` block:

```
<intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.BROWSABLE" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:scheme="geoname"/>
</intent-filter>
```

This intent filter definition uses a predefined action called `VIEW`, the action for viewing particular content. It is also handles Intent objects in the `BROWSABLE` or `DEFAULT` category and uses a scheme of `geoname` so that when a Uri starts with `geoname://`, the activity with this intent filter can be launched to view the content. You can read more

about this particular intent filter in *Android Wireless Application Development Volume II: Advanced Topics* when we discuss developing applications using location-based services (LBS).



### Tip

You can define custom actions unique to your application. If you do so, be sure to document these actions if you want them to be used by third parties. You can document these however you want: Your SDK documentation could be provided on your website, or confidential documents could be given directly to a client. For the most visibility, consider using an online registry, such as that at OpenIntents ([www.openintents.org](http://www.openintents.org)).

## Registering Other Application Components

All application components must be defined within the Android manifest file. In addition to activities, all services and broadcast receivers must be registered within the Android Manifest file.

- Services are registered using the `<service>` tag.
- Broadcast receivers are registered using the `<receiver>` tag.
- Content providers are registered using the `<provider>` tag.

Both services and broadcast receivers use intent filters. You learn much more about services, broadcast receivers, and intent filters in *Android Wireless Application Development Volume II: Advanced Topics*.

If your application acts as a content provider, effectively exposing a shared data service for use by other applications, it must declare this capability within the Android manifest file using the `<provider>` tag. Configuring a content provider involves determining what subsets of data are shared and what permissions are required to access them, if any. We begin our discussion of content providers in Chapter 14, “Using Content Providers.” You learn how to develop your own content providers in the second volume of this book.

## Working with Permissions

The Android operating system has been locked down so that applications have limited capability to adversely affect operations outside their process space. Instead, Android applications run within the bubble of their own virtual machine, with their own Linux user account (and related permissions).

### Registering Permissions Your Application Requires

Android applications have no permissions by default. Instead, any permissions for shared resources or privileged access—whether it’s shared data, such as the Contacts database, or

access to underlying hardware, such as the built-in camera—must be explicitly registered within the Android manifest file. These permissions are granted when the application is installed.



### Tip

When users install the application, they are informed what permissions the application requires to run and must approve these permissions. Request only the permissions your application requires.

The following XML excerpt for the preceding Android manifest file defines a permission using the `<uses-permission>` tag to gain access to the built-in camera:

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

A complete list of the permissions can be found in the `android.Manifest.permission` class. Your application manifest should include only the permissions required to run. The user is informed what permissions each Android application requires at install time.



### Tip

You might find that, in certain cases, permissions are not enforced (you can operate without the permission) by one device or another. In these cases, it is prudent to request the permission anyway for two reasons. First, the user is informed that the application is performing those sensitive actions, and, second, that permission could be enforced in a later device update. Also be aware that in early SDK versions, not all permissions were necessarily enforced at the platform level.



### Warning

Be aware that users will see these permissions before they install your application. If the application description or type of application you are providing does not clearly justify the permissions requested, you may get low ratings simply for asking for unnecessary permissions.

## Registering Permissions Your Application Enforces

Applications can also define and enforce their own permissions via the `<permission>` tag to be used by other applications. Permissions must be described and then applied to specific application components, such as activities, using the `android:permission` attribute.



### Tip

Use Java-style scoping for unique naming of application permissions (for example, `com.androidbook.MultiMedia.ViewMatureMaterial`).

Permissions can be enforced at several points:

- When starting an `Activity` or `Service`
- When accessing data provided by a content provider
- At the function call level
- When sending or receiving broadcasts by an `Intent`

Permissions can have three primary protection levels: `normal`, `dangerous`, and `signature`. The `normal` protection level is a good default for fine-grained permission enforcement within the application. The `dangerous` protection level is used for higher-risk activities, which might adversely affect the device. Finally, the `signature` protection level permits any application signed with the same certificate to use that component for controlled application interoperability. You learn more about application signing in Chapter 19.

Permissions can be broken down into categories, called `permission groups`, which describe or warn why specific activities require permission. For example, permissions might be applied for activities that expose sensitive user data such as location and personal information (`android.permission-group.LOCATION` and `android.permission-group.PERSONAL_INFO`), access underlying hardware (`android.permission-group.HARDWARE_CONTROLS`), or perform operations that might incur fees to the user (`android.permission-group.COST MONEY`). A complete list of permission groups is available within the `Manifest.permission_group` class.

For more information about applications and how they can enforce their own permissions, check out the `<permission>` manifest tag SDK documentation at <http://d.android.com/guide/topics/manifest/permission-element.html>.

## Exploring Other Manifest File Settings

We have now covered the basics of the Android manifest file, but many other settings are configurable within the Android manifest file using different tag blocks, not to mention attributes within each tag we already discussed.

Some other features you can configure within the Android manifest file include

- Setting application-wide themes within the `<application>` tag attributes
- Configuring unit-testing features using the `<instrumentation>` tag
- Aliasing activities using the `<activity-alias>` tag
- Creating broadcast receivers using the `<receiver>` tag
- Creating content providers using the `<provider>` tag, along with managing content provider permissions using the `<grant-uri-permission>` and `<path-permission>` tags
- Including other data within your activity, service, or receiver component registrations with the `<meta-data>` tag

For more detailed descriptions of each tag and attribute available in the Android SDK (and there are many), please review the Android SDK reference on the Android manifest file at <http://d.android.com/guide/topics/manifest/manifest-intro.html>.

## Summary

Each Android application has a specially formatted XML configuration file called `AndroidManifest.xml`. This file describes the application's identity in great detail. Some information you must define within the Android manifest file includes the application's name and version information, what application components it contains, which device configurations it requires, and what permissions it needs to run. The Android manifest file is used by the Android operating system to install, upgrade, and run the application package. Some details of the Android manifest file are also used by third parties, including the Android Market publication channel.

## References and More Information

Android Dev Guide: “The `AndroidManifest.xml` File”:

<http://d.android.com/guide/topics/manifest/manifest-intro.html>

Android Dev Guide: “Android API Levels”:

<http://d.android.com/guide/appendix/api-levels.html>

Android Dev Guide: “Supporting Multiple Screens”:

[http://d.android.com/guide/practices/screens\\_support.html](http://d.android.com/guide/practices/screens_support.html)

Android Dev Guide: “Security and Permissions”:

<http://d.android.com/guide/topics/security/security.html>

# Managing Application Resources

The well-written application accesses its resources programmatically instead of the developer hard-coding them into the source code. This is done for a variety of reasons. Storing application resources in a single place is a more organized approach to development and makes the code more readable and maintainable. Externalizing resources such as strings makes it easier to localize applications for different languages and geographic regions. Finally, different resources may be necessary for different devices.

In this chapter, you learn how Android applications store and access important resources such as strings, graphics, and other data. You also learn how to organize Android resources within the project files for localization and different device configurations.

## What Are Resources?

All Android applications are composed of two things: functionality (code instructions) and data (resources). The functionality is the code that determines how your application behaves. This includes any algorithms that make the application run. Resources include text strings, images and icons, audio files, videos, and other data used by the application.



### Tip

Many of the code examples provided in this chapter are taken from the SimpleResourceView, ResourceRoundup, and ParisView applications. The source code for these applications is provided for download on the book's website.

## Storing Application Resources

Android resource files are stored separately from the .java class files in the Android project. Most common resource types are stored in XML. You can also store raw data files and graphics as resources. Resources are organized in a strict directory hierarchy.

All resources must be stored under the `/res` project directory in specially named subdirectories that must be lowercase.

Different resource types are stored in different directories. The resource subdirectories generated when you create an Android project are shown in Table 7.1.

Table 7.1 Default Android Resource Directories

Resource Subdirectory	Purpose
<code>/res/drawable-*/</code>	Graphics resources
<code>/res/layout/</code>	User interface resources
<code>/res/values/</code>	Simple data such as strings and color values

Each resource type corresponds to a specific resource subdirectory name. For example, all graphics are stored under the `/res/drawable` directory structure. Resources can be further organized in a variety of ways using even more specially named directory qualifiers. For example, the `/res/drawable-hdpi` directory stores graphics for high-density screens, the `/res/drawable-ldpi` directory stores graphics for low-density screens, and the `/res/drawable-mdpi` directory stores graphics for medium-density screens. If you had a graphic resource that was shared by all screens, you would simply store that resource in the `/res/drawable` directory. We talk more about resource directory qualifiers later in this chapter.

If you use Eclipse with the Android Development Tools (ADT) plug-in, you will find that adding resources to your project is simple. The plug-in automatically detects new resources when you add them to the appropriate project resource subdirectory under `res`. These resources are compiled, resulting in the generation of the `R.java` source file, which enables you to access your resources programmatically.

## Resource Value Types

Android applications rely on many different types of resources—such as text strings, graphics, and color schemes—for user interface design.

These resources are stored in the `/res` directory of your Android project in a strict (but reasonably flexible) set of directories and files. All resource filenames must be lowercase and simple (letters, numbers, and underscores only).

The resource types supported by the Android SDK and how they are stored within the project are shown in Table 7.2.

Table 7.2 How Common Resource Types are Stored in the Project File Hierarchy

Resource Type	Required Directory	Suggested Filenames	XML Tag
Strings	/res/values/	strings.xml	<string>
String Pluralization	/res/values/	strings.xml	<plurals>, <item>
Arrays of Strings	/res/values/	strings.xml or arrays.xml	<string-array>, <item>
Booleans	/res/values/	bools.xml	<bool>
Colors	/res/values/	colors.xml	<color>
Color State Lists	/res/color/	Examples include buttonstates.xml, indicators.xml	<selector>, <item>
Dimensions	/res/values/	dimens.xml	<dimen>
Integers	/res/values/	integers.xml	<integer>
Arrays of Integers	/res/values/	integers.xml	<integer-array>
Mixed-Type Arrays	/res/values/	arrays.xml	<array>, <item>
Simple Drawables (Paintable)	/res/values/	drawables.xml	<drawable>
Graphics	/res/drawable/	Examples include icon.png, logo.jpg	Supported graphics files or drawable defi- nition XML files such as shapes
Tweened Animations	/res/anim/	Examples include fadesequence.xml, spinsequence.xml	<set>, <alpha>, <scale>, <translate>, <rotate>
Property Animations	/res/animator/	Examples include mypropanims.xml	<set>, <objectAnimator>, <valueAnimator>
Frame-by-Frame Animations	/res/drawable/	Examples include sequence1.xml, sequence2.xml	<animation-list>, <item>
Menus	/res/menu/	Examples include mainmenu.xml, helpmenu.xml	<menu>
XML Files	/res/xml/	Examples include data.xml, data2.xml	Defined by the developer.

Table 7.2 Continued

Resource Type	Required Directory	Suggested Filenames	XML Tag
Raw Files	/res/raw/	Examples include jingle.mp3, somevideo.mp4, helptext.txt	Defined by the developer.
Layouts	/res/layout/	Examples include main.xml, help.xml	Varies. Must be a layout control.
Styles and Themes	/res/values/	styles.xml, themes.xml	<style>



### Tip

Some resource files, such as animation files and graphics, are referenced by variables named from their filename (regardless of file suffix), so name your files appropriately. New resource types are added frequently to the Android SDK. Check the Android Developer website for more details at <http://goo.gl/WkSKr>.

## Storing Primitive Resource Types

Simple resource value types, such as strings, colors, dimensions, and other primitives, are stored under the /res/values project directory in XML files. Each resource file under the /res/values directory should begin with the following XML header:

```
<?xml version="1.0" encoding="utf-8"?>
```

Next comes the root node `<resources>` followed by the specific resource element types such as `<string>` or `<color>`. Each resource is defined using a different element name. Primitive resource types simply have a unique name and a value, like this color resource:

```
<color name="myFavoriteShadeOfRed">#800000</color>
```



### Tip

Although the XML filenames are arbitrary, the best practice is to store your resources in separate files to reflect their types, such as `strings.xml`, `colors.xml`, and so on. However, there's nothing stopping the developers from creating multiple resource files for a given type, such as two separate XML files called `bright_colors.xml` and `muted_colors.xml`, if they so choose. You will learn later in Chapter 15, “Designing Compatible Applications,” about alternative resources, too, that influence how the files may be named and subdivided.

## Storing Graphics and Files

In addition to simple resource types stored in the `/res/values` directory, you can also store numerous other types of resources, such as graphics, arbitrary XML files, and raw files. These types of resources are not stored in the `/res/values` directory, but instead are stored in specially named directories according to their type. For example, graphics are stored as files in the `/res/drawable` directory structure. XML files can be stored in the `/res/xml` directory and raw files can be stored in the `/res/raw` directory.

Make sure you name resource files appropriately because the resource name for graphics and files is derived from the filename of the specific resource. For example, a file called `flag.png` in the `/res/drawable` directory is given the name `R.drawable.flag`.

## Storing Other Resource Types

All other resource types—be they tweened animation sequences, color state lists, or menus—are stored in special XML formats in various directories, as discussed in Table 7.2. Again, each resource must be uniquely named.

## Understanding How Resources Are Resolved

The Android platform has a very robust mechanism for loading the appropriate resources at runtime. You can organize Android project resources based on more than a dozen different types of criteria. It can be useful to think of the resources stored at the directory hierarchy discussed in this chapter as the application’s *default resources*. You can also supply special versions of your resources to load instead of the defaults under certain conditions. These specialized resources are called *alternative resources*.

Two common reasons that developers use alternative resources are for internationalization and localization purposes and to design an application that runs smoothly on different device screens and orientations. We focus on default resources in this chapter and discuss alternative resources later in Chapter 15 as well as in *Android Wireless Application Development Volume II: Advanced Topics*.

Default and alternative resources are best illustrated by example. Let’s presume that we have a simple application with its requisite string, graphic, and layout resources. In this application, the resources are stored in the top-level resource directories (for example, `/res/values/strings.xml`, `/res/drawable/mylogo.png`, and `/res/layout/main.xml`). No matter what Android device (huge hi-def screen, postage-stamp-sized screen, portrait or landscape orientation, and so on) you run this application on, the same resource data is loaded and used. This application uses only default resources.

But what if we want our application to use different graphic sizes based on the screen density? We could use alternative graphic resources to do this. For example, we could provide different logos for different device screen densities by providing four versions of `mylogo.png`:

`/res/drawable-ldpi/mylogo.png` (low-density screens)

`/res/drawable-mdpi/mylogo.png` (medium-density screens)

```
/res/drawable-hdpi/mylogo.png (high-density screens)  
/res/drawable-xhdpi/mylogo.png (extra-high-density screens)
```

Let's look at another example. Let's say we find that the application would look much better if the layout were fully customized for portrait versus landscape orientations. We could change the layout around, moving controls around, in order to achieve a more pleasant user experience, and provide two layouts:

```
/res/layout-port/main.xml (layout loaded in portrait mode)  
/res/layout-land/main.xml (layout loaded in landscape mode)
```

We are introducing the concept of alternative resources now because they are hard to avoid completely, but we will work primarily with default resources for most of this book, simply in order to focus on specific programming tasks without the clutter that results from trying to customize an application to run beautifully on every device configuration one might use.

## Accessing Resources Programmatically

Developers access specific application resources using the `R.java` class file and its subclasses, which are automatically generated when you add resources to your project (if you use Eclipse). You can refer to any resource identifier in your project by its name (which is why it must be unique). For example, a string resource named `strHello` defined within the resource file called `/res/values/strings.xml` is accessed in the code as follows:

```
R.string.strHello
```

This variable is not the actual data associated with the string named hello. Instead, you use this resource identifier to retrieve the resource of that type (which happens to be string) from the project resources associated with the application.

First, you retrieve the `Resources` instance for your application `Context` (`android.content.Context`), which is, in this case, this because the `Activity` class extends `Context`. Then you use the `Resources` instance to get the appropriate kind of resource you want. You find that the `Resources` class (`android.content.res.Resources`) has helper methods for handling every kind of resource.

For example, a simple way to retrieve the string text is to call the `getString()` method of the `Resources` class, like this:

```
String myString = getResources().getString(R.string.strHello);
```

Before we go any further, we find it can be helpful to dig in and create some resources, so let's create a simple example.

## Setting Simple Resource Values Using Eclipse

To illustrate how to set resources using the ADT plug-in, let's look at an example. Create a new Android project and navigate to the /res/values/strings.xml file in Eclipse and double-click the file to edit it. Alternatively, you can use the Android project included with the book called ResourceRoundup to follow along. Your strings.xml resource file opens in the right pane and should look something like Figure 7.1, but with fewer strings.

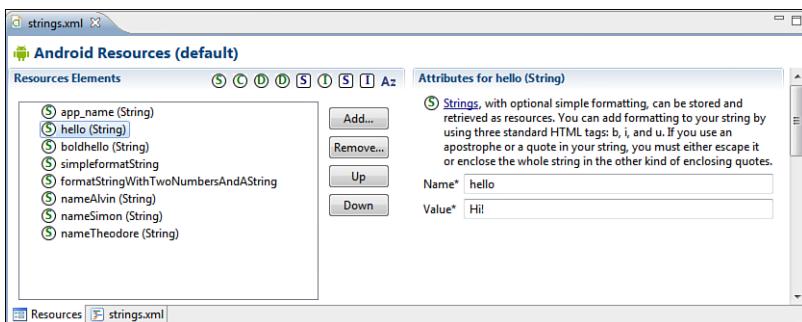


Figure 7.1 A sample string resource file in the Eclipse Resource Editor (Editor view).

There are two tabs at the bottom of this pane. The Resources tab provides a friendly method to easily insert primitive resource types such as strings, colors, and dimension resources. The strings.xml tab shows the raw XML resource file you are creating. Sometimes, editing the XML file manually is much faster, especially if you add a number of new resources. Click the strings.xml tab, and your pane should look something like Figure 7.2.

Now add some resources using the Add button on the Resources tab. Specifically, create the following resources:

- A color resource named prettyTextColor with a value of #ff0000
- A dimension resource named textSize with a value of 14pt
- A drawable resource named redDrawable with a value of #F00

Now you have several resources of various types in your strings.xml resource file. If you switch back to the XML view, you see that the Eclipse resource editor has added the appropriate XML elements to your file, which now should look something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<string name="app_name">ResourceRoundup</string>
<string>
```

```

        name="hello">Hello World, ResourceRoundupActivity</string>
<color name="prettyTextColor">#ff0000</color>
<dimen name="textPointSize">14pt</dimen>
<drawable name="redDrawable">#F00</drawable>
</resources>

```



Figure 7.2 A sample string resource file in the Eclipse Resource Editor (XML view).

Save the `strings.xml` resource file. The ADT plug-in automatically generates the `R.java` file in your project, with the appropriate resource IDs, which enable you to programmatically access your resources after they are compiled into the project. If you navigate to your `R.java` file, which is located under the `/src` directory in your package, it looks something like this:

```

package com.androidbook.resourceroundup;
public final class R {
    public static final class attr {
    }
    public static final class color {
        public static final int prettyTextColor=0x7f050000;
    }
    public static final class dimen {
        public static final int textSize=0x7f060000;
    }
    public static final class drawable {
        public static final int icon=0x7f020000;
        public static final int redDrawable=0x7f020001;
    }
    public static final class layout {
        public static final int main=0x7f030000;
    }
    public static final class string {
        public static final int app_name=0x7f040000;
    }
}

```

```
    public static final int hello=0x7f040001;
}
}
```

Now you are free to use these resources in your code. If you navigate to your `ResourceRoundupActivity.java` source file, you can add some lines to retrieve your resources and work with them, like this:

```
String myString = getResources().getString(R.string.hello);
int myColor =
    getResources().getColor(R.color.prettyTextColor);
float myDimen =
    getResources().getDimension(R.dimen.textPointSize);
ColorDrawable myDraw = (ColorDrawable)(getResources() .
    getDrawable(R.drawable.redDrawable));
```

Some resource types, such as string arrays, are more easily added to resource files by editing the XML by hand. For example, if we go back to the `strings.xml` file and choose the `strings.xml` tab, we can add a string array to our resource listing by adding the following XML element:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Use Some Resources</string>
    <string
        name="hello">Hello World, UseSomeResources</string>
    <color name="prettyTextColor">#ff0000</color>
    <dimen name="textPointSize">14pt</dimen>
    <drawable name="redDrawable">#F00</drawable>
    <string-array name="flavors">
        <item>Vanilla</item>
        <item>Chocolate</item>
        <item>Strawberry</item>
    </string-array>
</resources>
```

Save the `strings.xml` file, and now this string array named “flavors” is available in your source file `R.java`, so you can use it programmatically in `ResourceRoundupActivity.java`, like this:

```
String[] aFlavors =
    getResources().getStringArray(R.array.flavors);
```

You now have a general idea how to add simple resources using the ADT plug-in, but there are quite a few different types of data available to add as resources. It is a common practice to store different types of resources in different files. For example, you might store the strings in `/res/values/strings.xml` but store the `prettyTextColor` color resource in `/res/values/colors.xml` and the `textPointSize` dimension resource in

/res/values/dimens.xml. Reorganizing where you keep your resources in the resource directory hierarchy does not change the names of the resources or the code used earlier to access the resources programmatically.

Now let's take a closer look at how to add some of the most common types of resources to your Android applications.

## Working with Different Types of Resources

In this section, we look at the specific types of resources available for Android applications, how they are defined in the project files, and how you can access this resource data programmatically.

For each type of resource, you learn what types of values can be stored and in what format. Some resource types (such as `String` and `Color`) are well supported with the ADT Plug-in Resource Editor, whereas others (such as `Animation` sequences) are more easily managed by editing the XML files directly.

### Working with String Resources

`String` resources are among the simplest resource types available to the developer. `String` resources might show text labels on form views and for help text. The application name is also stored as a `String` resource, by default.

`String` resources are defined in XML under the /res/values project directory and compiled into the application package at build time. All strings with apostrophes or single straight quotes need to be escaped or wrapped in double straight quotes. Some examples of well-formatted string values are shown in Table 7.3.

Table 7.3 String Resource Formatting Examples

String Resource Value	Displays As
Hello, World	Hello, World
"User's Full Name:"	User's Full Name:
User\'s Full Name:	User's Full Name:
She said, \"Hi.\\"	She said, "Hi."
She\'s busy but she did say, \"Hi.\\"	She's busy but she did say, "Hi."

You can edit the `strings.xml` file using the Resources tab, or you can edit the XML directly by clicking the file and choosing the `strings.xml` tab. After you save the file, the resource identifiers are automatically added to your `R.java` class file.

`String` values are appropriately tagged with the `<string>` tag and represent a name/value pair. The name attribute is how you refer to the specific string programmatically, so name these resources wisely.

Here's an example of the string resource file `/res/values/strings.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Resource Viewer</string>
    <string name="test_string">Testing 1,2,3</string>
    <string name="test_string2">Testing 4,5,6</string>
</resources>
```

### Bold, Italic, and Underlined Strings

You can also add three HTML-style attributes to `String` resources. These are bold, italic, and underlining. You specify the styling using the `<b>`, `<i>`, and `<u>` tags, respectively. For example:

```
<string
    name="txt"><b>Bold</b>,<i>Italic</i>,<u>Line</u></string>
```

## Using String Resources as Format Strings

You can create format strings, but you need to escape all bold, italic, and underlining tags if you do so. For example, this text shows a score and the “win” or “lose” string:

```
<string
    name="winLose">Score: %1$d of %2$d! You %3$s.</string>
```

If you want to include bold, italic, or underlining in this format string, you need to escape the format tags. For example, if want to italicize the “win” or “lose” string at the end, your resource would look like this:

```
<string name="winLoseStyled">
    Score: %1$d of %2$d! You &lt;i&gt;%3$s&lt;/i&gt;.</string>
```

### Note

Those of you familiar with XML will recognize this as standard XML escaping. Indeed, that's all it is. After the standard set of XML escape characters is parsed, the string is then interpreted with the formatting tags. As with any XML document, you'd also need to escape single quotes (' is `&apos;`), double quotes (" is `&quot;`), and ampersands (& is `&amp;`).

## Using String Resources Programmatically

As shown earlier in this chapter, accessing `String` resources in code is straightforward. There are two primary ways in which you can access a `String` resource.

The following code accesses your application's `String` resource named `hello`, returning only the string. All HTML-style attributes (bold, italic, and underlining) are stripped from the string.



```
String myStrHello =
    getResources().getString(R.string.hello);
```

You can also access the string and preserve the formatting by using this other method:

```
CharSequence myBoldStr =
    getResources().getText(R.string.boldhello);
```

To load a format string, you need to make sure any format variables are properly escaped. One way you can do this is by using the `htmlEncode()` method of the `TextUtils` (`android.text.TextUtils`) class:

```
String mySimpleWinString;
mySimpleWinString =
    getResources().getString(R.string.winLose);
String escapedWin = TextUtils.htmlEncode(mySimpleWinString);
String resultText = String.format(mySimpleWinString, 5, 5, escapedWin);
```

The resulting text in the `resultText` variable is

Score: 5 of 5! You Won.

Now if you have styling in this format string like the preceding `String` resource `winLoseStyled`, you need to take a few more steps to handle the escaped italic tags. For this, you might want to use the `fromHtml()` method of the `Html` class (`android.text.Html`), as shown here:

```
String myStyledWinString;
myStyledWinString =
    getResources().getString(R.string.winLoseStyled);
String escapedWin = TextUtils.htmlEncode(myStyledWinString);
String resultText =
    String.format(myStyledWinString, 5, 5, escapedWin);
CharSequence styledResults = Html.fromHtml(resultText);
```

The resulting text in the `styledResults` variable is

Score: 5 of 5! You <i>Won</i>.

This variable, `styledResults`, can then be used in user interface controls such as `TextView` objects, where styled text is displayed correctly.



### Tip

A special resource type called `<plurals>` can be used to define strings that change based on a singular or plural form. For example, you could define a plural for the related strings:

"You caught a goose!"  
and

"You caught %d geese!"

Pluralized strings are loaded using the `getQuantityString()` method of the `Resources` class instead of the `getString()` method. For more information, see the Android SDK documentation regarding the `plurals` element.

## Working with String Arrays

You can specify lists of strings in resource files. This can be a good way to store menu options and drop-down list values. String arrays are defined in XML under the `/res/values` project directory and compiled into the application package at build time.

String arrays are appropriately tagged with the `<string-array>` tag and a number of `<item>` child tags, one for each string in the array. Here's an example of a simple array resource file `/res/values/arrays.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="flavors">
        <item>Vanilla Bean</item>
        <item>Chocolate Fudge Brownie</item>
        <item>Strawberry Cheesecake</item>
        <item>Coffee, Coffee, Buzz Buzz Buzz</item>
        <item>Americone Dream</item>
    </string-array>
    <string-array name="soups">
        <item>Vegetable minestrone</item>
        <item>New England clam chowder</item>
        <item>Organic chicken noodle</item>
    </string-array>
</resources>
```

As shown earlier in this chapter, accessing `String` array resources is easy. The method `getStringArray()` retrieves a `String` array from a resource file, in this case one named `flavors`:

```
String[] aFlavors =
    getResources().getStringArray(R.array.flavors);
```

## Working with Boolean Resources

Other primitive types are supported by the Android resource hierarchy as well. Boolean resources can be used to store information about application game preferences and default values. Boolean resources are defined in XML under the `/res/values` project directory and compiled into the application package at build time.

## Defining Boolean Resources in XML

Boolean values are appropriately tagged with the `<bool>` tag and represent a name/value pair. The name attribute is how you refer to the specific Boolean value programmatically, so name these resources wisely.

Here's an example of the Boolean resource file `/res/values/bools.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <bool name="bOnePlusOneEqualsTwo">true</bool>
    <bool name="bAdvancedFeaturesEnabled">false</bool>
</resources>
```

## Using Boolean Resources Programmatically

To use a Boolean resource in code, you can load it using the `getBoolean()` method of the `Resources` class. The following code accesses your application's Boolean resource named `bAdvancedFeaturesEnabled`:

```
boolean bAdvancedMode =
    getResources().getBoolean(R.bool.bAdvancedFeaturesEnabled);
```

## Working with Integer Resources

In addition to strings and Boolean values, you can also store integers as resources. Integer resources are defined in XML under the `/res/values` project directory and compiled into the application package at build time.

## Defining Integer Resources in XML

Integer values are appropriately tagged with the `<integer>` tag and represent a name/value pair. The name attribute is how you refer to the specific integer programmatically, so name these resources wisely.

Here's an example of the integer resource file `/res/values/nums.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <integer name="numTimesToRepeat">25</integer>
    <integer name="startingAgeOfCharacter">3</integer>
</resources>
```

## Using Integer Resources Programmatically

To use the integer resource, you must load it using the `Resources` class. The following code accesses your application's integer resource named `numTimesToRepeat`:

```
int repTimes = getResources().getInteger(R.integer.numTimesToRepeat);
```



### Tip

Much like with string arrays, you can create integer arrays as resources using the `<integer-array>` tag with child `<item>` tags, defining one for each item in the array. You can then load the integer array using the `getIntArray()` method of the `Resources` class.

## Working with Colors

Android applications can store RGB color values, which can then be applied to other screen elements. You can use these values to set the color of text or other elements, such as the screen background. Color resources are defined in XML under the `/res/values` project directory and compiled into the application package at build time.

### Defining Color Resources in XML

RGB color values always start with the hash symbol (#). The alpha value can be given for transparency control. The following color formats are supported:

- #RGB (for example, #F00 is 12-bit color, red)
- #ARGB (for example, #8F00 is 12-bit color, red with alpha 50%)
- #RRGGBB (for example, #FF00FF is 24-bit color, magenta)
- #AARRGGBB (for example, #80FF00FF is 24-bit color, magenta, with alpha 50%)

Color values are appropriately tagged with the `<color>` tag and represent a name/value pair. Here's an example of a simple color resource file `/res/values/colors.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="background_color">#006400</color>
    <color name="text_color">#FFE4C4</color>
</resources>
```

### Using Color Resources Programmatically

The example at the beginning of the chapter accessed a color resource. Color resources are simply integers. The following example shows the method `getColor()` retrieving a color resource called `prettyTextColor`:

```
int myResourceColor =
    getResources().getColor(R.color.prettyTextColor);
```

## Working with Dimensions

Many user interface layout controls, such as text controls and buttons, are drawn to specific dimensions. These dimensions can be stored as resources. Dimension values always end with a unit of measurement tag.

### Defining Dimension Resources in XML

Dimension values are tagged with the `<dimen>` tag and represent a name/value pair. Dimension resources are defined in XML under the `/res/values` project directory and compiled into the application package at build time.

The dimension units supported are shown in Table 7.4.

Table 7.4 Dimension Unit Measurements Supported

Unit of Measurement	Description	Resource Tag Required	Example
Pixels	Actual screen pixels	px	20px
Inches	Physical measurement	in	1in
Millimeters	Physical measurement	mm	1mm
Points	Common font measurement unit	pt	14pt
Screen density-independent pixels	Pixels relative to 160dpi screen (preferable dimension for screen compatibility)	dp	1dp
Scale-independent pixels	Best for scalable font display	sp	14sp

Here's an example of a simple dimension resource file called `/res/values/dimens.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <dimen name="FourteenPt">14pt</dimen>
    <dimen name="OneInch">1in</dimen>
    <dimen name="TenMillimeters">10mm</dimen>
    <dimen name="TenPixels">10px</dimen>
</resources>
```



#### Note

Generally, `dp` is used for layouts and graphics whereas `sp` is used for text. A device's default settings will usually result in `dp` and `sp` being the same. However, because the user can control the size of text when it's in `sp` units, you would not use `sp` for text where the font layout size was important, such as with a title. Instead, it's good for content text where the users settings might be important (such as a really large font for the vision impaired).

## Using Dimension Resources Programmatically

Dimension resources are simply floating-point values. The `getDimension()` method retrieves a dimension resource called `textPointSize`:

```
float myDimension =  
    getResources().getDimension(R.dimen.textPointSize);
```

### Warning

Be cautious when choosing dimension units for your applications. If you are planning to target multiple devices, with different screen sizes and resolutions, then you need to rely heavily on the more scalable dimension units, such as `dp` and `sp`, as opposed to pixels, points, inches, and millimeters.



## Working with Simple Drawables

You can specify simple colored rectangles by using the `drawable` resource type, which can then be applied to other screen elements. These `drawable` types are defined in specific paint colors, much like the `color` resources are defined.

### Defining Simple Drawable Resources in XML

Simple paintable drawable resources are defined in XML under the `/res/values` project directory and compiled into the application package at build time. Paintable drawable resources use the `<drawable>` tag and represent a name/value pair. Here's an example of a simple drawable resource file called `/res/values/drawables.xml`:

```
<?xml version="1.0" encoding="utf-8"?>  
<resources>  
    <drawable name="red_rect">#F00</drawable>  
</resources>
```

Although it might seem a tad confusing, you can also create XML files that describe other `Drawable` subclasses, such as `ShapeDrawable`. `Drawable` XML definition files are stored in the `/res/drawable` directory within your project, along with image files. This is not the same as storing `<drawable>` resources, which are paintable drawables. `ShapeDrawable` resources are stored in the `/res/values` directory, as explained previously.

Here's a simple `ShapeDrawable` described in the file `/res/drawable/red_oval.xml`:

```
<?xml version="1.0" encoding="utf-8"?>  
<shape  
    xmlns:android=  
        "http://schemas.android.com/apk/res/android"  
    android:shape="oval">  
        <solid android:color="#f00"/>  
</shape>
```

Of course, we don't need to specify the size because it will scale automatically to the layout it's placed in, much like any vector graphics format. We talk more about graphics and drawing shapes in *Android Wireless Application Development Volume II: Advanced Topics*.

### Using Simple Drawable Resources Programmatically

Drawable resources defined with <drawable> are simply rectangles of a given color, which is represented by the Drawable subclass ColorDrawable. The following code retrieves a ColorDrawable resource called redDrawable:

```
ColorDrawable myDraw = (ColorDrawable) getResources().  
    getDrawable(R.drawable.redDrawable);
```



#### Tip

Many additional drawable resource types can be specified as XML resources. These special drawables correspond to specific drawable classes such as ClipDrawable and LevelListDrawable. For information on these specialized drawable types, see the Android SDK documentation.

## Working with Images

Applications often include visual elements such as icons and graphics. Android supports several image formats that can be directly included as resources for your application. These image formats are shown in Table 7.5.

Table 7.5 Image Formats Supported in Android

Supported Image Format	Description	Required Extension
Portable Network Graphics (PNG)	Preferred format (lossless)	.png
Nine-Patch Stretchable Images	Preferred format (lossless)	.9.png
Joint Photographic Experts Group (JPEG)	Acceptable format (lossy)	.jpg, .jpeg
Graphics Interchange Format (GIF)	Discouraged format	.gif

These image formats are all well supported by popular graphics editors such as Adobe Photoshop, GIMP, and Microsoft Paint. The Nine-Patch Stretchable Graphics can be created from PNG files using the `draw9patch` tool included with the Android SDK under the `/tools` directory. Adding image resources to your project is easy. Simply drag the image asset into the `/res/drawable` resource directory hierarchy and it will automatically be included in the application package.



#### Warning

All resources filenames must be lowercase and simple (letters, numbers, and underscores only). This rule applies to all files, including graphics.

## Working with Nine-Patch Stretchable Graphics

Android device screens, be they smartphones, tablets, or TVs, come in various dimensions. It can be handy to use stretchable graphics to allow a single graphic that can scale appropriately for different screen sizes and orientations or different lengths of text. This can save you or your designer a lot of time in creating graphics for many different screen sizes.

Android supports Nine-Patch Stretchable Graphics for this purpose. Nine-Patch graphics are simply PNG graphics that have patches, or areas of the image, defined to scale appropriately, instead of scaling the entire image as one unit. Often the center segment is transparent or a solid color for a background because it's the stretched part. As such, a common use for Nine-Patch graphics is to create frames and borders. Little more than the corners are needed, so a very small graphic file can be used to frame any sized image or view control.

Nine-Patch Stretchable Graphics can be created from PNG files using the `draw9patch` tool included with the `/tools` directory of the Android SDK. We talk more about compatibility and using Nine-Patch graphics in Chapter 15.

## Using Image Resources Programmatically

Image resources are simply another kind of `Drawable` called a `BitmapDrawable`. Most of the time, you need only the resource ID of the image to set as an attribute on a user interface control.

For example, if we drop the graphics file `flag.png` into the `/res/drawable` directory and add an `ImageView` control to the main layout, we can interact with that control programmatically in the layout by first using the `findViewById()` method to retrieve a control by its identifier and casting it to the proper type of control—in this case, an `ImageView` (`android.widget.ImageView`) object:

```
ImageView flagImageView =
    (ImageView) findViewById(R.id.ImageView01);
flagImageView.setImageResource(R.drawable.flag);
```

Similarly, if you want to access the `BitmapDrawable` (`android.graphics.drawable.BitmapDrawable`) object directly, you can request that resource directly using the `getDrawable()` method, as follows:

```
BitmapDrawable bitmapFlag = (BitmapDrawable)
    getResources().getDrawable(R.drawable.flag);
int iBitmapHeightInPixels =
    bitmapFlag.getIntrinsicHeight();
int iBitmapWidthInPixels = bitmapFlag.getIntrinsicWidth();
```

Finally, if you work with Nine-Patch graphics, the call to `getDrawable()` will return a `NinePatchDrawable` (`android.graphics.drawable.NinePatchDrawable`) object instead of a `BitmapDrawable` object:

```
NinePatchDrawable stretchy = (NinePatchDrawable)
    getResources().getDrawable(R.drawable.pyramid);
int iStretchyHeightInPixels =
    stretchy.getIntrinsicHeight();
int iStretchyWidthInPixels = stretchy.getIntrinsicWidth();
```



### Tip

A special resource type called `<selector>` can be used to define different colors or drawables to be used depending on a control's state. For example, you could define a color state list for a `Button` control: gray when the button is disabled, green when it is enabled, and yellow when it is being pressed. Similarly, you could provide different drawables based on the state of an `ImageButton` control. For more information, see the Android SDK documentation regarding the color and drawable state list resources.

## Working with Animation

Android supports several kinds of animation. Two of the simplest varieties are *frame-by-frame* animation and *tweening*. Frame-by-frame animation involves the display of a sequence of images in rapid succession. Tweened animation involves applying standard graphical transformations such as rotations and fades upon a single image.

The Android SDK provides some helper utilities for loading and using animation resources. These utilities are found in the `android.view.animation.AnimationUtils` class.

We discuss animation in detail in Volume II of this book series. For now, let's just look at how you define these animation types in terms of resources.

### Defining and Using Frame-by-Frame Animation Resources

Frame-by-frame animation is often used when the content changes from frame to frame. This type of animation can be used for complex frame transitions—much like a kid's flip-book.

To define frame-by-frame resources, take the following steps:

1. Save each frame graphic as an individual drawable resource. It may help to name your graphics sequentially, in the order in which they are displayed—for example, `frame1.png`, `frame2.png`, and so on.
2. Define the animation set resource in an XML file within the `/res/drawable/` resource directory hierarchy.
3. Load, start, and stop the animation programmatically.

Here's an example of a simple frame-by-frame animation resource file called `/res/drawable/juggle.xml` that defines a simple three-frame animation that takes 1.5 seconds to complete a single loop:

```
<?xml version="1.0" encoding="utf-8" ?>
<animation-list
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot="false">
    <item
        android:drawable="@drawable/splash1"
        android:duration="500" />
    <item
        android:drawable="@drawable/splash2"
        android:duration="500" />
    <item
        android:drawable="@drawable/splash3"
        android:duration="500" />
</animation-list>
```

Frame-by-frame animation set resources defined with `<animation-list>` are represented by the Drawable subclass `AnimationDrawable`. The following code retrieves an `AnimationDrawable` resource called `juggle`:

```
AnimationDrawable jugglerAnimation = (AnimationDrawable) getResources() .
    getDrawable(R.drawable.juggle);
```

After you have a valid `AnimationDrawable` (`android.graphics.drawable.AnimationDrawable`), you can assign it to a `View` control on the screen and start and stop animation.

## Defining and Using Tweened Animation Resources

Tweened animation features include scaling, fading, rotation, and translation. These actions can be applied simultaneously or sequentially and might use different interpolators.

Tweened animation sequences are not tied to a specific graphic file, so you can write one sequence and then use it for a variety of different graphics. For example, you can make moon, star, and diamond graphics all pulse using a single scaling sequence, or you can make them spin using a rotate sequence.

## Defining Tweened Animation Sequence Resources in XML

Graphic animation sequences can be stored as specially formatted XML files in the `/res/anim` directory and are compiled into the application binary at build time.

Here's an example of a simple animation resource file called `/res/anim/spin.xml` that defines a simple rotate operation—rotating the target graphic counterclockwise four times in place, taking 10 seconds to complete:

```
<?xml version="1.0" encoding="utf-8" ?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <rotate
        android:fromDegrees="0"
        android:toDegrees="-1440"
```

```

        android:pivotX="50%"
        android:pivotY="50%"
        android:duration="10000" />
</set>
```

## Using Tweened Animation Sequence Resources Programmatically

If we go back to the earlier example of a `BitmapDrawable`, we can now include some animation simply by adding the following code to load the animation resource file `spin.xml` and set the animation in motion:

```

ImageView flagImageView =
    (ImageView) findViewById(R.id.ImageView01);
flagImageView.setImageResource(R.drawable.flag);
...
Animation an =
    AnimationUtils.loadAnimation(this, R.anim.spin);
flagImageView.startAnimation(an);
```

Now you have your graphic spinning. Notice that we loaded the animation using the base class object `Animation`. You can also extract specific animation types using the subclasses that match: `RotateAnimation`, `ScaleAnimation`, `TranslateAnimation`, and `AlphaAnimation` (found in the `android.view.animation` package).

There are a number of different interpolators you can use with your tweened animation sequences.

## Working with Menus

You can also include menu resources in your project files. Like animation resources, menu resources are not tied to a specific control but can be reused in any menu control.

### Defining Menu Resources in XML

Each menu resource (which is a set of individual menu items) is stored as a specially formatted XML file in the `/res/menu` directory and are compiled into the application package at build time.

Here's an example of a simple menu resource file called `/res/menu/speed.xml` that defines a short menu with four items in a specific order:

```

<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/start"
        android:title="Start!"
        android:orderInCategory="1"></item>
    <item
        android:id="@+id/stop" android:title="Stop!"
        android:orderInCategory="4"></item>
    <item
```

```
    android:id="@+id/accel"
    android:title="Vroom! Accelerate!"
    android:orderInCategory="2"></item>
<item
    android:id="@+id/decel"
    android:title="Decelerate!"
    android:orderInCategory="3"></item>
</menu>
```

You can create menus using the ADT plug-in, which can access the various configuration attributes for each menu item. In the previous case, we set the title (label) of each menu item and the order in which the items display. Now, you can use string resources for those titles instead of typing in the strings. For example:

```
<menu xmlns:android=
    "http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/start"
        android:title="@string/start"
        android:orderInCategory="1"></item>
    <item
        android:id="@+id/stop"
        android:title="@string/stop"
        android:orderInCategory="2"></item>
</menu>
```

## Using Menu Resources Programmatically

To access the preceding menu resource called `/res/menu/speed.xml`, simply override the method `onCreateOptionsMenu()` in your `Activity` class, returning `true` to cause the menu to be displayed:

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

That's it. Now if you run your application and press the menu button, you see the menu. A number of other XML attributes can be assigned to menu items. For a complete list of these attributes, see the Android SDK reference for menu resources at the website <http://d.android.com/guide/topics/resources/menu-resource.html>. You learn a lot more about menus and menu event handling in Chapter 8, “Exploring User Interface Screen Elements.”

## Working with XML Files

You can include arbitrary XML resource files to your project. You should store these XML files in the `/res/xml` directory, and they are compiled into the application package at build time.

The Android SDK has a variety of packages and classes available for XML manipulation. You learn more about XML handling in Chapter 13, “Working with Files and Directories.” For now, we create an XML resource file and access it through code.

## Defining Raw XML Resources

First, put a simple XML file in the /res/xml directory. In this case, the file `my_pets.xml` with the following contents can be created:

```
<?xml version="1.0" encoding="utf-8"?>
<pets>
    <pet name="Bit" type="Bunny" />
    <pet name="Nibble" type="Bunny" />
    <pet name="Stack" type="Bunny" />
    <pet name="Queue" type="Bunny" />
    <pet name="Heap" type="Bunny" />
    <pet name="Null" type="Bunny" />
    <pet name="Nigiri" type="Fish" />
    <pet name="Sashimi II" type="Fish" />
    <pet name="Kiwi" type="Lovebird" />
</pets>
```

## Using XML Resources Programmatically

Now you can access this XML file as a resource programmatically in the following manner:

```
XmlResourceParser myPets =
    getResources().getXml(R.xml.my_pets);
```

You can then use the parser of your choice to parse the XML. We discuss working with files, including XML files, in Chapter 13, “Working with Files and Directories.”

## Working with Raw Files

Your application can also include raw files as part of its resources. For example, your application might use raw files such as audio files, video files, and other file formats not supported by the Android Resource packaging tool `aapt`.

### Defining Raw File Resources

All raw resource files are included in the /res/raw directory and are added to your package without further processing.



#### Warning

All resources filenames must be lowercase and simple (letters, numbers, and underscores only). This also applies to raw file filenames even though the tools do not process these files other than to include them in your application package.

The resource filename must be unique to the directory and should be descriptive because the filename (without the extension) becomes the name by which the resource is accessed.

### Using Raw File Resources Programmatically

You can access raw file resources from the `/res/raw` resource directory and any resource from the `/res/drawable` directory (bitmap graphics files, anything not using the `<resource>` XML definition method). Here's one way to open a file called `the_help.txt`:

```
InputStream iFile =  
    getResources().openRawResource(R.raw.the_help);
```

## References to Resources

You can reference resources instead of duplicating them. For example, your application might want to reference a single string resource in multiple string arrays.

The most common use of resource references is in layout XML files, where layouts can reference any number of resources to specify attributes for layout colors, dimensions, strings, and graphics. Another common use is within style and theme resources.

Resources are referenced using the following format:

```
@resource_type/variable_name
```

Recall that earlier we had a string array of soup names. If we want to localize the soup listing, a better way to create the array is to create individual string resources for each soup name and then store the references to those string resources in the string array (instead of the text).

To do this, we define the string resources in the `/res/values/strings.xml` file like this:

```
<?xml version="1.0" encoding="utf-8"?>  
<resources>  
    <string name="app_name">Application Name</string>  
    <string name="chicken_soup">Organic Chicken Noodle</string>  
    <string name="minestrone_soup">Veggie Minestrone</string>  
    <string name="chowder_soup">New England Lobster Chowder</string>  
</resources>
```

And then we can define a localizable string array that references the string resources by name in the `/res/values/arrays.xml` file like this:

```
<?xml version="1.0" encoding="utf-8"?>  
<resources>  
    <string-array name="soups">  
        <item>@string/minestrone_soup</item>  
        <item>@string/chowder_soup</item>  
        <item>@string/chicken_soup</item>  
    </string-array>  
</resources>
```



### Tip

Save the `strings.xml` file first so that the string resources (which are picked up by the `aapt` and included in the `R.java` class) are defined prior to trying to save the `arrays.xml` file, which references those particular string resources. Otherwise, you might get the following error:

```
Error: No resource found that matches the given name.
```

You can also use references to make aliases to other resources. For example, you can alias the system resource for the OK string to an application resource name by including the following in your `strings.xml` resource file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string id="app_ok">@android:string/ok</string>
</resources>
```

You learn more about all the different system resources available later in this chapter.



### Tip

Much like with string and integer arrays, you can create arrays of any type of resources using the `<array>` tag with child `<item>` tags, defining one item for each resource in the array. You can then load the array of miscellaneous resources using the `obtainTypedArray()` method of the `Resources` class. The typed array resource is commonly used for grouping and loading a bunch of `Drawable` resources with a single call. For more information, see the Android SDK documentation on typed array resources.

## Working with Layouts

Much as web designers use HTML, user interface designers can use XML to define Android application screen elements and layout. A layout XML resource is where many different resources come together to form the definition of an Android application screen. Layout resource files are included in the `/res/layout/` directory and are compiled into the application package at build time. Layout files might include many user interface controls and define the layout for an entire screen or describe custom controls used in other layouts.

Here's a simple example of a layout file (`/res/layout/main.xml`) that sets the screen's background color and displays some text in the middle of the screen (see Figure 7.3).

The `main.xml` layout file that displays this screen references a number of other resources, including colors, strings, and dimension values, all of which were defined in the `strings.xml`, `colors.xml`, and `dimens.xml` resource files. The color resource for the screen background color and resources for a `TextView` control's color, string, and text size follow:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="@color/background_color">
    <TextView
        android:id="@+id/TextView01"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:text="@string/test_string"
        android:textColor="@color/text_color"
        android:gravity="center"
        android:textSize="@dimen/text_size" />
</LinearLayout>
```

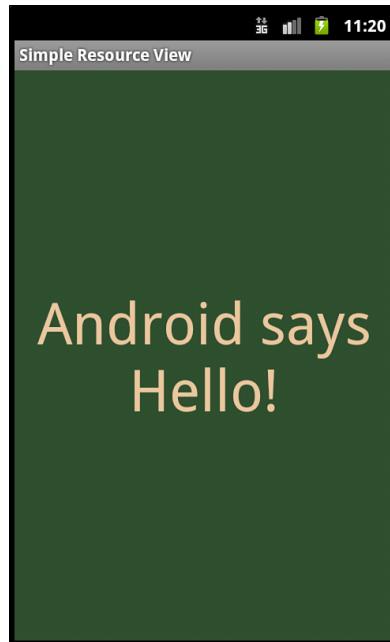


Figure 7.3 How the main.xml layout file displays in the emulator.

The preceding layout describes all the visual elements on a screen. In this example, a `LinearLayout` control is used as a container for other user interface controls—here, a single `TextView` that displays a line of text.



### Tip

You can encapsulate common layout definitions in their own XML files and then include those layouts within other layout files using the `<include>` tag. For example, you can use the following `<include>` tag to include another layout file called `/res/layout/mygreenrect.xml` within the `main.xml` layout definition:

```
<include layout="@layout/mygreenrect"/>
```

## Designing Layouts in Eclipse

Layouts can be designed and previewed in Eclipse using the Resource editor functionality provided by the ADT plug-in (see Figure 7.4). If you click the project file `/res/layout/main.xml` (provided with any new Android project), you see the Layout tab, which shows you a preview of the layout, and the `main.xml` tab, which shows you the raw XML of the layout file.

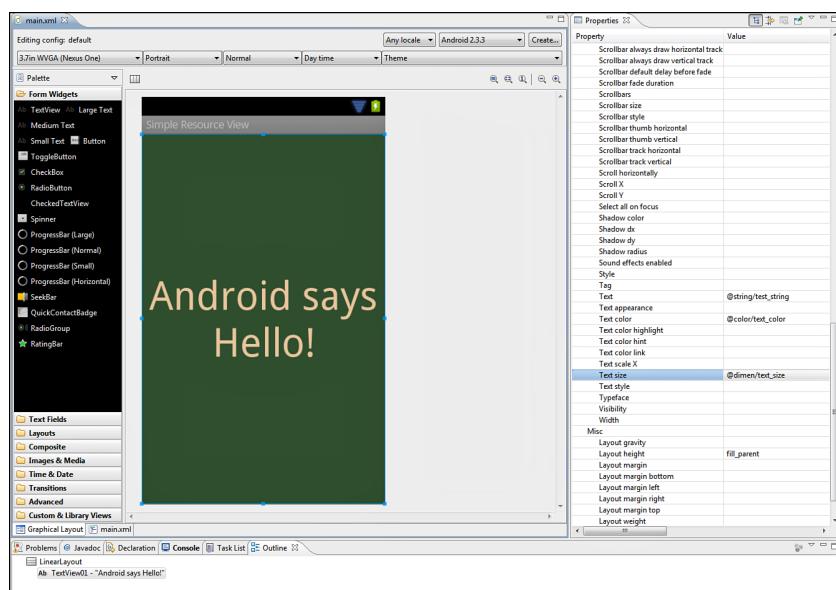


Figure 7.4 Designing a layout file using Eclipse.

As with most user interface editors, the ADT plug-in works well for your basic layout needs, enables you to create user interface controls such as `TextView` and `Button` controls easily, and enables setting the controls' properties in the Properties pane.



### Tip

Moving the Properties pane to the far right of the workspace in Eclipse makes it easier to browse and set control properties when designing layouts.

Now is a great time to get to know the layout resource editor, also called the layout designer. Try creating a new Android project called ParisView (available as a sample project). Navigate to the `/res/layout/main.xml` layout file and double-click it to open it in the editor. It's quite simple by default, with only a black (empty) rectangle and a string of text.

Below in the Resource pane of the Eclipse perspective, you notice the Outline tab. This outline is the XML hierarchy of this layout file. By default, you see a `LinearLayout`. If you expand it, you see it contains one `TextView` control. Click the `TextView` control. You see that the Properties pane of the Eclipse perspective now has all the properties available for that object. If you scroll down to the property called `text`, you see that it's set to the string resource variable `@string/hello`.



### Tip

You can also select specific controls by clicking them in the layout designer's preview area. The currently selected control is highlighted in red. We prefer to use the Outline view, so we can be sure we are clicking what we expect.

You can use the layout designer to set and preview layout control properties. For example, you can modify the `TextView` property called `textSize` by typing `18pt` (a dimension). You see the results of your change to the property immediately in the preview area.

Take a moment to switch to the `main.xml` tab. You notice that the properties you set are now in the XML. If you save and run your project in the emulator now, you see similar results to what you see in the designer preview.

Now go back to the Outline pane. You see a green plus button and a red minus button. You can use these buttons to add and remove controls to your layout file. For example, select the `LinearLayout` from the Outline view and then click the green button to add a control within that container object.

Choose the `ImageView` object. Now you have a new control in your layout. You can't actually see it yet because it is not fully defined.

Drag two PNG graphics files (or JPGs) into your `/res/drawable` project directory, naming them `flag.png` and `background.png`. Now, browse the properties of your `ImageView` control and then set the `src` property by clicking the resource browser button labeled [...]. You can browse all the `Drawable` resources in your project and select the flag resource you just added. You can also set this property manually by typing `@drawable/flag`.

Now, you see that the graphic shows up in your preview. While we're at it, select the `LinearLayout` object and set its `background` property to the background Drawable you added.

If you save the layout file and run the application in the emulator (as shown in Figure 7.5) or on the phone, you see results much like you did in the resource designer preview pane.



Figure 7.5 A layout with a `LinearLayout`, `TextView`, and `ImageView`, shown in the Android emulator.

### Using Layout Resources Programmatically

Objects within layouts, whether they are `Button` or `ImageView` controls, are all derived from the `View` class. Here's how you would retrieve a `TextView` object named `TextView01`, called in an `Activity` class after the call to `setContentView()`:

```
TextView txt = (TextView) findViewById(R.id.TextView01);
```

You can also access the underlying XML of a layout resource much as you would any XML file. The following code retrieves the `main.xml` layout file for XML parsing:

```
XmlResourceParser myMainXml =  
    getResources().getLayout(R.layout.main);
```

Developers can also define custom layouts with unique attributes. We talk much more about layout files and designing Android user interfaces in Chapter 9, “Designing User Interfaces with Layouts.”

### Warning

The Java code associated with your project is pretty much unaware of which version of a resource is loaded—whether it's the default or some alternative version. Take special care when providing alternative layout resources. Layout resources tend to be more complicated; the child controls within them are often referred to in code by name. Therefore, if you begin to create alternative layout resources, make sure each named child control that is referenced in code exists in each alternative layout. For example, if you have a user interface with a `Button` control, make sure the `Button` control's identifier (`android:id`) is the same in both the landscape, portrait, and other alternative layout resources. You may include different controls and properties in each layout and rearrange them as you like, but those controls that are referred to and interacted with programmatically should exist in all layouts so that your code runs smoothly, regardless of the layout loaded. If they don't, you'll need to code conditionally, or you may even want to consider whether the screen is so different it should be represented by a different `Activity` class.

## Referencing System Resources

In addition to the resources included in your project, you can also take advantage of the generic resources provided as part of the Android SDK. You can access system resources much as you would your own resources. The `android` package contains all kinds of resources, which you can browse by looking in the `android.R` subclasses. Here you find system resources for

- Animation sequences for fading in and out
- Arrays of email/phone types (home, work, and such)
- Standard system colors
- Dimensions for application thumbnails and icons
- Many commonly used `drawable` and layout types
- Error strings and standard button text
- System styles and themes

You can reference system resources in other resources such as layout files by specifying the @android package name before the resource. For example, to set the background to the system color for darker gray, you set the appropriate background color attribute to @android:color/darr\_gray.

You can access system resources programmatically through the android.R class. If we go back to our animation example, we could have used a system animation instead of defining our own. Here is the same animation example again, except it uses a system animation to fade in:

```
ImageView flagImageView =  
    (ImageView) findViewById(R.id.ImageView01);  
flagImageView.setImageResource(R.drawable.flag);  
Animation an = AnimationUtils.  
    loadAnimation(this, android.R.anim.fade_in);  
flagImageView.startAnimation(an);
```



### Warning

Although referencing system resources can be useful to give your application a more consistent look with the rest of a particular device's user interface (something users will appreciate), you still need to be cautious when using them. If a particular device has system resources that are dramatically different, or fail to include specific resources your application relies upon, then your application may not look right or behave as expected. An installable application, called rs:ResEnum (<http://goo.gl/lr3V8>), can be used to enumerate and display the various system resources available on a given device. Thus, you can quickly verify system resource availability across your target devices.

## Summary

Android applications rely on various types of resources, including strings, string arrays, colors, dimensions, drawable objects, graphics, animation sequences, layouts, and more. Resources can also be raw files. Many of these resources are defined with XML and organized into specially named project directories. Both default and alternative resources can be defined using this resource hierarchy.

Resources are compiled and accessed using the R.java class file, which is automatically generated by the ADT plug-in for Eclipse when the application resources are saved, allowing developers to access the resources programmatically.

## References and More Information

Android Dev Guide: “Application Resources”:

<http://d.android.com/guide/topics/resources/index.html>

Android Dev Guide: “Resource Types”:

<http://d.android.com/guide/topics/resources/available-resources.html>



# Android User Interface Design Essentials

- 8** Exploring User Interface Screen Elements
- 9** Designing User Interfaces with Layouts
- 10** Working with Fragments
- 11** Working with Dialogs

*This page intentionally left blank*

# Exploring User Interface Screen Elements

Most Android applications inevitably need some form of user interface. In this chapter, we discuss the user interface elements available within the Android Software Development Kit (SDK). Some of these elements display information to the user, whereas others are input controls that can be used to gather information from the user. In this chapter, you learn how to use a variety of common user interface controls to build different types of screens.

## Introducing Android Views and Layouts

Before we go any further, we need to define a few terms. This gives you a better understanding of certain capabilities provided by the Android SDK before they are fully introduced. First, let's talk about the `View` and what it is to the Android SDK.

### Introducing the Android View

The Android SDK has a Java package named `android.view`. This package contains a number of interfaces and classes related to drawing on the screen. However, when we refer to the `View` object, we actually refer to only one of the classes within this package: the `android.view.View` class.

The `View` class is the basic user interface building block within Android. It represents a rectangular portion of the screen. The `View` class serves as the base class for nearly all the user interface controls and layouts within the Android SDK.

### Introducing the Android Controls

The Android SDK contains a Java package named `android.widget`. When we refer to controls, we are typically referring to a class within this package. The Android SDK includes classes to draw most common objects, including `ImageView`, `FrameLayout`, `EditText`, and `Button` classes. As mentioned previously, all controls are typically derived from the `View` class.

This chapter is primarily about controls that display and collect data from the user. We cover many of these basic controls in detail.

Your layout resource files are composed of different user interface controls. Some are static, and you don't need to work with them programmatically. Others you'll want to be able to access and modify in your Java code. Each control you want to be able to access programmatically must have a unique identifier specified using the `android:id` attribute. You use this identifier to access the control with the `findViewById()` method in your `Activity` class. Most of the time, you'll want to cast the `View` returned to the appropriate control type. For example, the following code illustrates how to access a `TextView` control using its unique identifier:

```
TextView tv = (TextView) findViewById(R.id.TextView01);
```



#### Note

Do not confuse the user interface controls in the `android.widget` package with App Widgets. An `AppWidget` (`android.appwidget`) is an application extension, often displayed on the Android Home screen. We discuss App Widgets in more depth in the second volume of this book.

## Introducing the Android Layout

One special type of control found within the `android.widget` package is called a layout. A layout control is still a `View` object, but it doesn't actually draw anything specific on the screen. Instead, it is a parent container for organizing other controls (children). Layout controls determine how and where on the screen child controls are drawn. Each type of layout control draws its children using particular rules. For instance, the `LinearLayout` control draws its child controls in a single horizontal row or a single vertical column. Similarly, a `TableLayout` control displays each child control in tabular format (in cells within specific rows and columns).

In Chapter 9, “Designing User Interfaces with Layouts,” we organize various controls within layouts and other containers. These special `View` controls, which are derived from the `android.view.ViewGroup` class, are useful only after you understand the various display controls these containers can hold. By necessity, we use some of the layout `View` objects within this chapter to illustrate how to use the controls previously mentioned. However, we don't go into the details of the various layout types available as part of the Android SDK until the next chapter.



#### Note

Many of the code examples provided in this chapter are taken from the `ViewSamples` application. The source code for the `ViewSamples` application is provided for download on the book's websites.

## Displaying Text to Users with TextView

One of the most basic user interface elements, or controls, in the Android SDK is the `TextView` control. You use it, quite simply, to draw text on the screen. You primarily use it to display fixed text strings or labels.

Frequently, the `TextView` control is a child control within other screen elements and controls. As with most of the user interface elements, it is derived from `View` and is within the `android.widget` package. Because it is a `View`, all the standard attributes such as width, height, padding, and visibility can be applied to the object. However, because this is a text-displaying control, you can apply many other `TextView` attributes to control behavior and how the text is viewed in a variety of situations.

First, though, let's see how to put some quick text up on the screen. `<TextView>` is the XML layout file tag used to display text on the screen. You can set the `android:text` property of the `TextView` to be either a raw text string in the layout file or a reference to a string resource.

Here are examples of both methods you can use to set the `android:text` attribute of a `TextView`. The first method sets the text attribute to a raw string; the second method uses a string resource called `sample_text`, which must be defined in the `strings.xml` resource file.

```
<TextView  
    android:id="@+id/TextView01"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Some sample text here" />  
  
<TextView  
    android:id="@+id/TextView02"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="@string/sample_text" />
```

To display this `TextView` on the screen, all your `Activity` needs to do is call the `setContentView()` method with the layout resource identifier where you defined the preceding XML shown. You can change the text displayed programmatically by calling the `setText()` method on the `TextView` object. Retrieving the text is done with the `getText()` method.

Now let's talk about some of the more common attributes of `TextView` objects.

## Configuring Layout and Sizing

The `TextView` control has some special attributes that dictate how the text is drawn and flows. You can, for instance, set the `TextView` to be a single line high and a fixed width. If, however, you put a long string of text that can't fit, the text truncates abruptly. Luckily, there are some attributes that can handle this problem.



### Tip

When looking through the attributes available to `TextView` objects, you should be aware that the `TextView` class contains all the functionality needed by editable controls. This means that many of the attributes apply only to input fields, which are used primarily by the subclass `EditText` object. For example, the `autoText` attribute, which helps the user by fixing common spelling mistakes, is most appropriately set on editable text fields (`EditText`). There is no need to use this attribute normally when you are simply displaying text.

The width of a `TextView` can be controlled in terms of the `ems` measurement rather than in pixels. An **em** is a term used in typography that is defined in terms of the point size of a particular font. (For example, the measure of an em in a 12-point font is 12 points.) This measurement provides better control over how much text is viewed, regardless of the font size. Through the `ems` attribute, you can set the width of the `TextView`. Additionally, you can use the `maxEms` and `minEms` attributes to set the maximum width and minimum width, respectively, of the `TextView` in terms of `ems`.

The height of a `TextView` can be set in terms of lines of text rather than pixels. Again, this is useful for controlling how much text can be viewed regardless of the font size. The `lines` attribute sets the number of lines that the `TextView` can display. You can also use `maxLines` and `minLines` to control the maximum height and minimum height, respectively, that the `TextView` displays.

Here is an example that combines these two types of sizing attributes. This `TextView` is two lines of text high and 12 `ems` of text wide. The layout width and height are specified to the size of the `TextView` and are required attributes in the XML schema:

```
<TextView  
    android:id="@+id/TextView04"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:lines="2"  
    android:ems="12"  
    android:text="@string/autolink_test" />
```

Instead of having the text only truncate at the end, as happens in the preceding example, we can enable the `ellipsize` attribute to replace the last couple characters with an ellipsis (...) so the user knows that not all text is displayed.

## Creating Contextual Links in Text

If your text contains references to email addresses, web pages, phone numbers, or even street addresses, you might want to consider using the attribute `autoLink` (see Figure 8.1). The `autoLink` attribute has four values that you can use in combination with each other. When enabled, these `autoLink` attribute values create standard web-style links to the application that can act on that data type. For instance, setting the attribute to `web` automatically finds and links any URLs to web pages.

Your text can contain the following values for the `autoLink` attribute:

- **none**: Disables all linking.
- **web**: Enables linking of URLs to web pages.
- **email**: Enables linking of email addresses to the mail client with the recipient filled.
- **phone**: Enables linking of phone numbers to the dialer application with the phone number filled out, ready to be dialed.
- **map**: Enables linking of street addresses to the map application to show the location.
- **all**: Enables all types of linking.

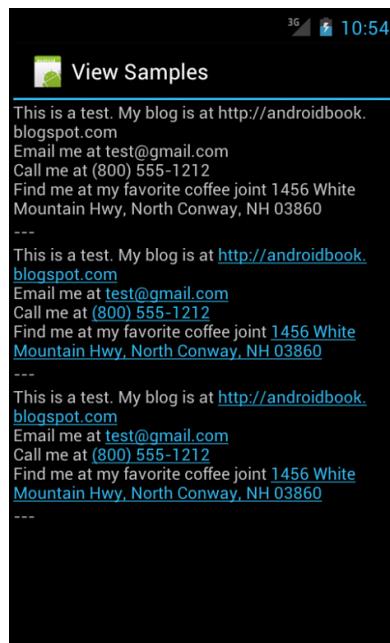


Figure 8.1 Three `TextView` types: Simple, AutoLink All (not clickable), and AutoLink All (clickable).

Turning on the `autoLink` feature relies on the detection of the various types within the Android SDK. In some cases, the linking might not be correct or might be misleading.

Here is an example that links email and web pages, which, in our opinion, are the most reliable and predictable:

```
<TextView
    android:id="@+id/TextView02"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/autolink_test"
    android:autoLink="web|email" />
```

Two helper values are available for this attribute as well. You can set it to none to make sure no type of data is linked. You can also set it to all to have all known types linked. Figure 8.2 illustrates what happens when you click these links. The default for a TextView is not to link any types. If you want the user to see the various data types highlighted but you don't want the user to click them, you can set the linksClickable attribute to false.

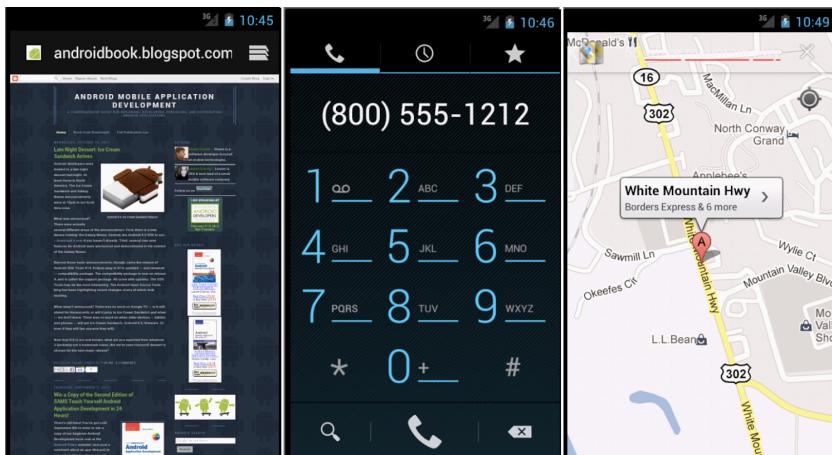


Figure 8.2 Clickable AutoLinks: a URL launches the browser, a phone number launches the dialer, and a street address launches Google Maps.

## Retrieving Data from Users with EditText

The Android SDK provides a number of controls for retrieving data from users. One of the most common types of data that applications often need to collect from users is text. Two frequently used controls to handle this type of job are `EditText` controls and `Spinner` controls (Android's version of a drop-down control).

### Retrieving Text Input Using EditText Controls

The Android SDK provides a convenient control called `EditText` to handle text input from a user. The `EditText` class is derived from `TextView`. In fact, most of its

functionality is contained within `TextView` but is enabled when created as an `EditText`. The `EditText` object has a number of useful features enabled by default, many of which are shown in Figure 8.3.

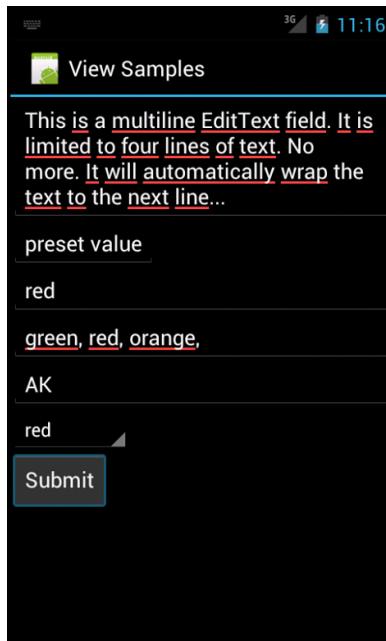


Figure 8.3 Various styles of `EditText`, `Spinner`, and `Button` controls.

First, though, let's see how to define an `EditText` control in an XML layout file:

```
<EditText  
    android:id="@+id/EditText01"  
    android:layout_height="wrap_content"  
    android:hint="type here"  
    android:lines="4"  
    android:layout_width="match_parent" />
```

This layout code shows a basic `EditText` element. There are a couple of interesting things to note. First, the `hint` attribute puts some text in the edit box that goes away when the user starts entering text (run the sample code to see an example of a hint in action). Essentially, this gives a hint to the user as to what should go there. Next is the `lines` attribute, which defines how many lines tall the input box is. If this is not set, the entry field grows as the user enters text. However, setting a size allows the user to scroll within a fixed sized to edit the text. This also applies to the width of the entry.

By default, the user can perform a long press to bring up a context menu. This provides to the user some basic copy, cut, and paste operations as well as the ability to change the input method and add a word to the user's dictionary of frequently used words (shown in Figure 8.4). You do not need to provide any additional code for this useful behavior to benefit your users. You can also highlight a portion of the text from code. A call to `setSelection()` does this, and a call to `selectAll()` highlights the entire text-entry field.

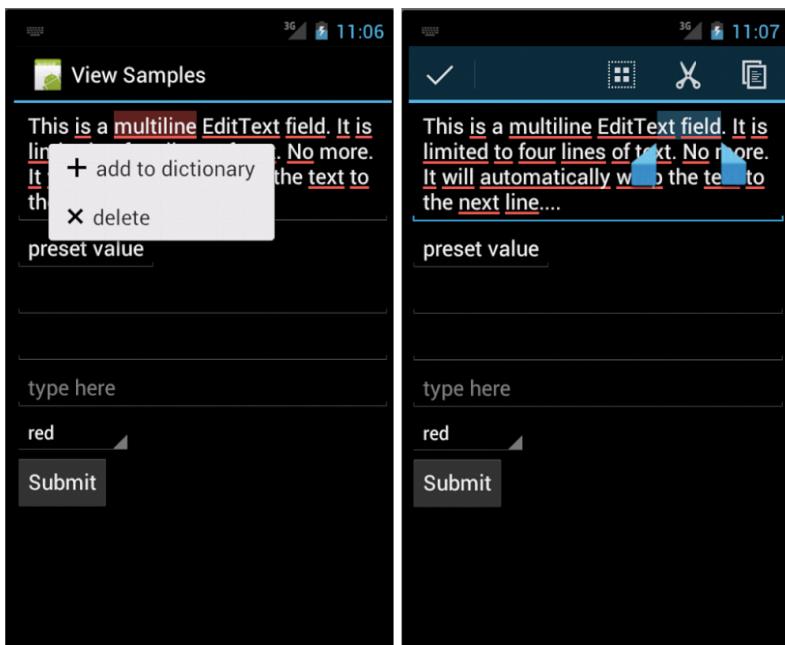


Figure 8.4 A long press on a `EditText` control typically launches a context menu for selections, cutting, and copying. (The Paste option appears when you have copied text.)

The `EditText` object is essentially an editable `TextView`. This means that you can read text from it in the same way as you did with `TextView`: by using the `getText()` method. You can also set initial text to draw in the text-entry area using the `setText()` method.

## Constraining User Input with Input Filters

There are times when you don't want the user to type just anything. Validating input after the user has entered something is one way to do this. However, a better way to

avoid wasting the user's time is to filter the input. The `EditText` control provides a way to set an `InputFilter` that does only this.

The Android SDK provides some `InputFilter` objects for use. `InputFilter` objects enforce such rules as allowing only uppercase text and limiting the length of the text entered. You can create custom filters by implementing the `InputFilter` interface, which contains the single method called `filter()`. Here is an example of an `EditText` control with two built-in filters that might be appropriate for a two-letter state abbreviation:

```
final EditText text_filtered =
    (EditText) findViewById(R.id.input_filtered);
text_filtered.setFilters(new InputFilter[] {
    new InputFilter.AllCaps(),
    new InputFilter.LengthFilter(2)
});
```

The `setFilters()` method call takes an array of `InputFilter` objects. This is useful for combining multiple filters, as shown. In this case, we convert all input to uppercase. Additionally, we set the maximum length to two characters long. The `EditText` control looks the same as any other, but if you try to type in lowercase, the text is converted to uppercase, and the string is limited to two characters. This does not mean that all possible inputs are valid, but it does help users to not concern themselves with making the input too long or bother with the case of the input. This also helps your application by guaranteeing that any text from this input is a length of two characters. It does not constrain the input to only letters, though.

## Helping the User with Autocompletion

In addition to providing a basic text editor with the `EditText` control, the Android SDK also provides a way to help the user with entering commonly used data into forms. This functionality is provided through the autocomplete feature.

There are two forms of autocomplete. One is the more standard style of filling in the entire text entry based on what the user types. If the user begins typing a string that matches a word in a developer-provided list, the user can choose to complete the word with just a tap. This is done through the `AutoCompleteTextView` control (see Figure 8.5, left). The second method allows the user to enter a list of items, each of which has autocomplete functionality (see Figure 8.5, right). These items must be separated in some way by providing a `Tokenizer` to the `MultiAutoCompleteTextView` object that handles this method. A common `Tokenizer` implementation is provided for comma-separated lists and is used by specifying the `MultiAutoCompleteTextView.CommaTokenizer` object. This can be helpful for lists of specifying common tags and the like.

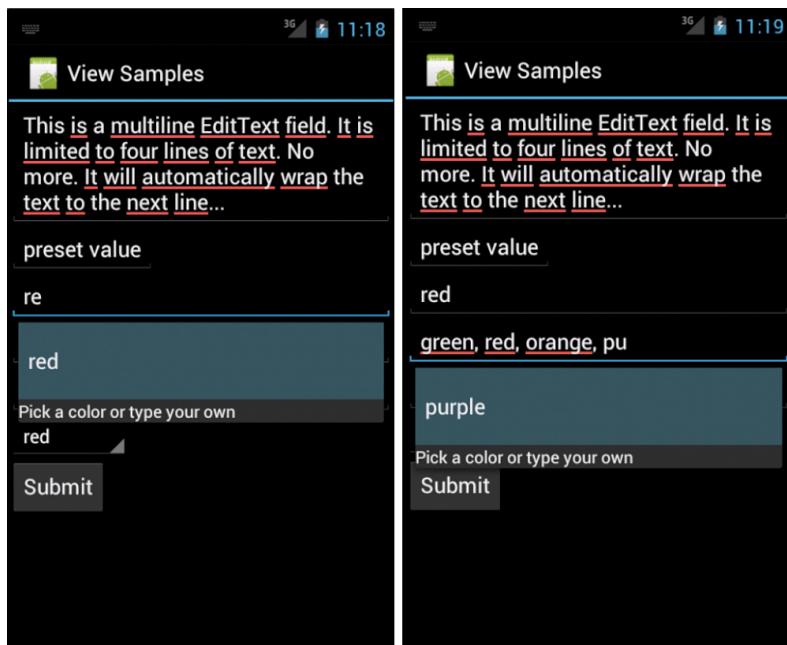


Figure 8.5 Using AutoCompleteTextView (left) and MultiAutoCompleteTextView (right).

Both of the autocomplete text editors use an adapter to get the list of text they use to provide completions to the user. This example shows how to provide an AutoCompleteTextView that can help users type some of the basic colors from an array in the code:

```
final String[] COLORS = {  
    "red", "green", "orange", "blue", "purple",  
    "black", "yellow", "cyan", "magenta" };  
ArrayAdapter<String> adapter =  
    new ArrayAdapter<String>(this,  
        android.R.layout.simple_dropdown_item_1line,  
        COLORS);  
AutoCompleteTextView text = (AutoCompleteTextView)  
    findViewById(R.id.AutoCompleteTextView01);  
text.setAdapter(adapter);
```

In this example, when the user starts typing in the field, if he starts with one of the letters in the COLORS array, a drop-down list shows all the available completions. Note that this does not limit what the user can enter. The user is still free to enter any text (such as “puce”). The adapter controls the look of the drop-down list. In this case, we use a

built-in layout made for such things. Here is the layout resource definition for this AutoCompleteTextView control:

```
<AutoCompleteTextView  
    android:id="@+id/AutoCompleteTextView01"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:completionHint="Pick a color or type your own"  
    android:completionThreshold="1" />
```

There are a couple more things to notice here. First, you can choose when the completion drop-down list shows by filling in a value for the completionThreshold attribute. In this case, we set it to a single character, so it displays immediately if there is a match. The default value is two characters of typing before it displays autocompletion options. Second, you can set some text in the completionHint attribute. This displays at the bottom of the drop-down list to help users. Finally, the drop-down list for completions is sized to the TextView. This means that it should be wide enough to show the completions and the text for the completionHint attribute.

The MultiAutoCompleteTextView is essentially the same as the regular autocomplete, except that you must assign a Tokenizer to it so that the control knows where each autocompletion should begin. The following is an example that uses the same adapter as the previous example but includes a Tokenizer for a list of user color responses, each separated by a comma:

```
MultiAutoCompleteTextView mtext =  
    (MultiAutoCompleteTextView) findViewById(R.id.MultiAutoCompleteTextView01);  
mtext.setAdapter(adapter);  
mtext.setTokenizer(new MultiAutoCompleteTextView.CommaTokenizer());
```

As you can see, the only change is setting the Tokenizer. Here, we use the built-in comma Tokenizer provided by the Android SDK. In this case, whenever a user chooses a color from the list, the name of the color is completed, and a comma is automatically added so that the user can immediately start typing in the next color. As before, this does not limit what the user can enter. If the user enters “maroon” and places a comma after it, the autocompletion starts again as the user types another color, regardless of the fact that it didn’t help the user type in the color maroon. You can create your own Tokenizer by implementing the MultiAutoCompleteTextView.Tokenizer interface. You can do this if you’d prefer entries separated by a semicolon or some other more complex separator.

## Giving Users Choices Using Spinner Controls

Sometimes you want to limit the choices available for users to type. For instance, if users are going to enter the name of a state, you might as well limit them to only the valid states, because this is a known set. Although you could do this by letting them type

something and then blocking invalid entries, you can also provide similar functionality with a `Spinner` control. As with the autocomplete method, the possible choices for a spinner can come from an `Adapter`. You can also set the available choices in the layout definition by using the `entries` attribute with an array resource (specifically a string array that is referenced as something such as `@array/state-list`). The `Spinner` control isn't actually an `EditText`, although it is frequently used in a similar fashion. Here is an example of the XML layout definition for a `Spinner` control for choosing a color:

```
<Spinner  
    android:id="@+id/Spinner01"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:entries="@array/colors"  
    android:prompt="@string/spin_prompt" />
```

This places a `Spinner` control on the screen. A closed `Spinner` control is shown in Figure 8.5, with just the first choice, red, displayed. An open `Spinner` control is shown in Figure 8.6, which shows all the color selections available. When the user selects this control, a pop-up shows the prompt text followed by a list of the possible choices. This list allows only a single item to be selected at a time, and when one is selected, the pop-up goes away.

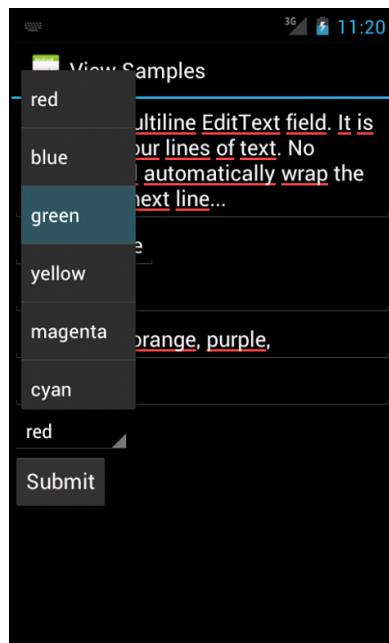


Figure 8.6 Filtering choices with a `Spinner` control.

There are a couple of things to notice here. First, the `entries` attribute is set to the value of a string array resource, referred to here as `@array/colors`. Second, the `prompt` attribute is defined as a string resource. Unlike some other string attributes, this one is required to be a string resource. The `prompt` displays when the `Spinner` control is opened and all selections are displayed. The `prompt` can be used to tell the user what kinds of values can be selected.

Because the `Spinner` control is not a `TextView`, but a list of `TextView` objects, you can't directly request the selected text from it. Instead, you have to retrieve the specific selected option (each of which is a `TextView` control) and extract the text directly from it:

```
final Spinner spin = (Spinner) findViewById(R.id.Spinner01);
TextView text_sel = (TextView) spin. getSelectedView();
String selected_text = text_sel.getText().toString();
```

Alternatively, we could have called the `getSelectedItem()` or `getSelectedItemID()` method to deal with other forms of selection.

## Allowing Simple User Selections with Buttons, Check Boxes, Switches, and Radio Groups

Another common user interface element is the button. In this section, you learn about different kinds of buttons provided by the Android SDK. These include the basic `Button`, `CheckBox`, `ToggleButton`, and `RadioButton`.

- A basic `Button` is often used to perform some sort of action, such as submitting a form or confirming a selection. A basic `Button` control can contain a text or image label.
- A `CheckBox` is a button with two states—checked and unchecked. You often use `CheckBox` controls to turn a feature on or off or to pick multiple items from a list.
- A `ToggleButton` is similar to a `CheckBox`, but you use it to visually show the state. The default behavior of a toggle is like that of a power on/off button.
- A `Switch` is similar to a `CheckBox`, in that it is a two-state control. The default behavior of a control is like a slider switch that can be moved between an “on” and “off” position. This control was introduced in API Level 14 (Android 4.0).
- A `RadioButton` provides selection of an item. Grouping `RadioButton` controls together in a container called a `RadioGroup` enables the developer to enforce that only one `RadioButton` is selected at a time.

You can find examples of each type of control in Figure 8.7.

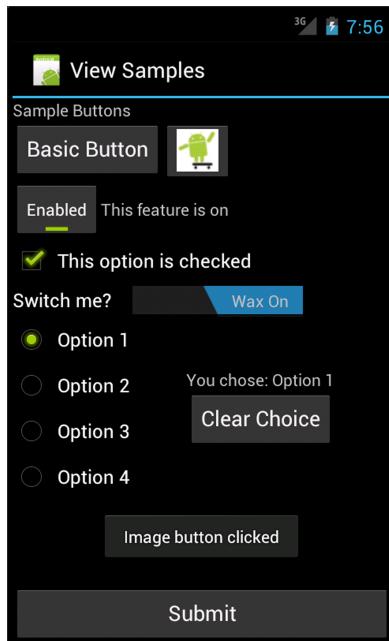


Figure 8.7 Various types of button controls.

## Using Basic Buttons

The `android.widget.Button` class provides a basic button implementation in the Android SDK. Within the XML layout resources, buttons are specified using the `Button` element. The primary attribute for a basic button is the `text` field. This is the label that appears on the middle of the button's face. You often use basic `Button` controls for buttons with text such as “Ok,” “Cancel,” or “Submit.”



### Tip

You can find many common application string values in the Android system resource `strings`, exposed in `android.R.string`. There are strings for common button text such as “yes,” “no,” “ok,” “cancel,” and “copy.” For more information on system resources, see Chapter 7, “Managing Application Resources.”

The following XML layout resource file shows a typical `Button` control definition:

```
<Button  
    android:id="@+id/basic_button"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Basic Button" />
```

A `Button` control won't do anything, other than animate, without some code to handle the click event. Here is an example of some code that handles a click for a basic button and displays a `Toast` message on the screen:

```
setContentView(R.layout.buttons);  
final Button basic_button = (Button) findViewById(R.id.basic_button);  
basic_button.setOnClickListener(new View.OnClickListener() {  
    public void onClick(View v) {  
        Toast.makeText(ButtonsActivity.this,  
            "Button clicked", Toast.LENGTH_SHORT).show();  
    }  
});
```

### Tip

A `Toast` (`android.widget.Toast`) is a simple dialog-like message that displays for a second or so and then disappears. `Toast` messages are useful for providing the user with nonessential confirmation messages; they are also quite handy for debugging. Figure 8.7 shows an example of a `Toast` message that displays the text "Image button clicked."

To handle the click event for when a `Button` control is pressed, we first get a reference to the `Button` by its resource identifier. Next, the `setOnClickListener()` method is called. It requires a valid instance of the class `View.OnClickListener`. A simple way to provide this is to define the instance right in the method call. This requires implementing the `onClick()` method. Within the `onClick()` method, you are free to carry out whatever actions you need. Here, we simply display a message to the users telling them that the button was, in fact, clicked.

A button-like control with its primary label as an image is an `ImageButton`. An `ImageButton` is, for most purposes, almost exactly like a basic button. Click actions are handled in the same way. The primary difference is that you can set its `src` attribute to be an image. Here is an example of an `ImageButton` definition in an XML layout resource file:

```
<ImageButton  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:id="@+id/image_button"  
    android:src="@drawable/droid" />
```

In this case, a small drawable resource is referenced. Refer to Figure 8.7 to see what this “Android” button looks like. (It’s to the right of the basic Button.)



### Tip

You can also use the `onClick` XML attribute to set the name of your click method within your `Activity` class, and implement it that way. Simply specify the name of your `Activity` class’s click method name using this attribute and define a `public void` method that takes a single `View` parameter and implement your click handling.

## Using CheckBox and ToggleButton Controls

The check box button is often used in lists of items where the user can select multiple items. The Android check box contains a `text` attribute that appears to the side of the check box. Because the `Checkbox` class is derived from the `TextView` and `Button` classes, much of the attributes and methods behave in a similar fashion.

Here’s an XML layout resource definition for a simple `CheckBox` control with some default text displayed:

```
<CheckBox  
    android:id="@+id/checkbox"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Check me?" />
```

The following example shows how to check for the state of the button programmatically and change the text label to reflect the change:

```
final CheckBox check_button = (CheckBox) findViewById(R.id.checkbox);  
check_button.setOnClickListener(new View.OnClickListener() {  
    public void onClick (View v) {  
        CheckBox cb = (CheckBox) findViewById(R.id.checkbox);  
        cb.setText(check_button.isChecked() ?  
            "This option is checked" :  
            "This option is not checked");  
    }  
});
```

This is similar to the basic `Button` control. A `CheckBox` control automatically shows the state as checked or unchecked. This enables us to deal with behavior in our application rather than worrying about how the button should behave. The layout shows that the text starts out one way, but after the user clicks the button the text changes to one of two different things, depending on the checked state. You can see how this `CheckBox` is displayed once it has been clicked (and the text has been updated) in Figure 8.7 (center).

A `ToggleButton` is similar to a check box in behavior but is usually used to show or alter the on or off state of something. Like the `CheckBox`, it has a state (checked or not).

Also like the check box, the act of changing what displays on the button is handled for us. Unlike the `CheckBox`, it does not show text next to it. Instead, it has two text fields. The first attribute is `textOn`, which is the text that displays on the button when its checked state is on. The second attribute is `textOff`, which is the text that displays on the button when its checked state is off. The default text for these is “ON” and “OFF”, respectively.

The following layout code shows a definition for a `ToggleButton` control that shows “Enabled” or “Disabled” based on the state of the button:

```
<ToggleButton  
    android:id="@+id/toggle_button"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Toggle"  
    android:textOff="Disabled"  
    android:textOn="Enabled" />
```

This type of button does not actually display the value for the `text` attribute, even though it’s a valid attribute to set. Here, the only purpose it serves is to demonstrate that it doesn’t display. You can see what this `ToggleButton` looks like in Figure 8.7 (“Enabled”).

The `Switch` control (`android.widget.Switch`), which was introduced in API Level 14, provides similar two-state behavior to the `ToggleButton` control, only instead of the control being clicked to toggle between the states, it looks more like a slider. The following layout code shows a definition for a `Switch` control with a prompt (“Switch Me?”) and two states: “Wax On” and “Wax Off”:

```
<Switch android:id="@+id/switch1" android:layout_width="wrap_content"  
       android:layout_height="wrap_content" android:text="Switch me?"  
       android:textOn="Wax On" android:textOff="Wax Off" />
```

## Using RadioGroup and RadioButton

You often use radio buttons when a user should be allowed to select only one item from a small group of items. For instance, a question asking for gender can give three options: male, female, and unspecified. Only one of these options should be checked at a time. The `RadioButton` objects are similar to `CheckBox` objects. They have a text label next to them, set via the `text` attribute, and they have a state (checked or unchecked). However, you can group `RadioButton` objects inside a `RadioGroup` that handles enforcing their combined states so that only one `RadioButton` can be checked at a time. If the user selects a `RadioButton` that is already checked, it does not become unchecked. However, you can provide the user with an action to clear the state of the entire `RadioGroup` so that none of the buttons are checked.

Here we have an XML layout resource with a `RadioGroup` containing four `RadioButton` objects (shown in Figure 8.7, toward the bottom of the screen). The

RadioButton objects have text labels, “Option 1,” “Option 2,” and so on. The XML layout resource definition is shown here:

```
<RadioGroup
    android:id="@+id/RadioGroup01"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">
    <RadioButton
        android:id="@+id/RadioButton01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Option 1" />
    <RadioButton
        android:id="@+id/RadioButton02"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Option 2" />
    <RadioButton
        android:id="@+id/RadioButton03"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Option 3" />
    <RadioButton
        android:id="@+id/RadioButton04"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Option 4" />
</RadioGroup>
```

You can handle actions on these RadioButton objects through the RadioGroup object. The following example shows registering for clicks on the RadioButton objects within the RadioGroup and setting the text of a TextView called TextView01, which is defined elsewhere in the layout file..

```
final RadioGroup group = (RadioGroup) findViewById(R.id.RadioGroup01);
final TextView tv = (TextView)
    findViewById(R.id.TextView01);

group.setOnCheckedChangeListener(new
    RadioGroup.OnCheckedChangeListener() {
    public void onCheckedChanged(
        RadioGroup group, int checkedId) {
        if (checkedId != -1) {
            RadioButton rb = (RadioButton)
                findViewById(checkedId);
            if (rb != null) {
                tv.setText("You chose: " + rb.getText());
            }
        }
    }
});
```

```
        }
    } else {
        tv.setText("Choose 1");
    }
}
});
```

As this layout example demonstrates, there is nothing special you need to do to make the `RadioGroup` and internal `RadioButton` objects work properly. The preceding code illustrates how to register to receive a notification whenever the `RadioButton` selection changes.

The code demonstrates that the notification contains the resource identifier for the specific `RadioButton` chosen by the user, as defined in the layout resource file. To do something interesting with this, you need to provide a mapping between this resource identifier (or the text label) and the corresponding functionality in your code. In the example, we query for the button that was selected, get its text, and assign its text to another `TextView` control that we have on the screen.

As mentioned, the entire `RadioGroup` can be cleared so that none of the `RadioButton` objects are selected. The following example demonstrates how to do this in response to a button click outside of the `RadioGroup`:

```
final Button clear_choice = (Button) findViewById(R.id.Button01);
clear_choice.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        RadioGroup group = (RadioGroup)
            findViewById(R.id.RadioGroup01);
        if (group != null) {
            group.clearCheck();
        }
    }
})
```

The action of calling the `clearCheck()` method triggers a call to the `onCheckedChangedListener()` callback method. This is why we have to make sure that the resource identifier we received is valid. Right after a call to the `clearCheck()` method, it is not a valid identifier but instead is set to the value `-1` to indicate that no `RadioButton` is currently checked.

### Tip

You can also handle `RadioButton` clicks using specific click handlers on individual `RadioButtons` within a `RadioGroup`. The implementation mirrors that of a regular `Button` control.

## Retrieving Dates and Times from Users

The Android SDK provides a couple controls for getting date and time input from the user. The first is the `DatePicker` control (see Figure 8.8, top). It can be used to get a month, day, and year from the user.

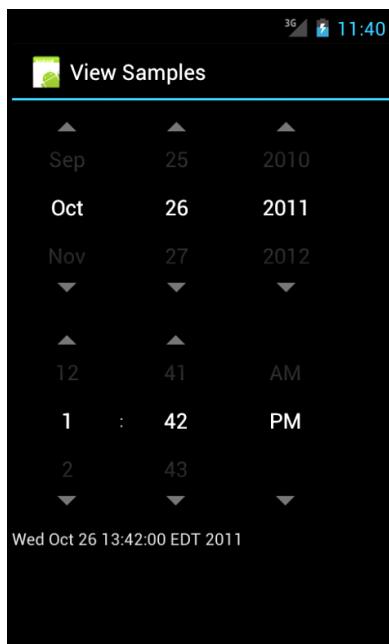


Figure 8.8 Date and time controls.

The basic XML layout resource definition for a `DatePicker` follows:

```
<DatePicker  
    android:id="@+id/DatePicker01"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:calendarViewShown="false"  
    android:spinnersShown="true" >  
</DatePicker>
```

As you can see from this example, a couple of attributes help control the look of the picker. Setting the `calendarViewShown` attribute to `true`, when using API Level 11 and up, will show a full calendar, including week numbers, but may take up more space than you can allow. Try it in the sample code, though, to see what it looks like. As with many of the other controls, your code can register to receive a method call when the date changes. You do this by implementing the `onDateChanged()` method:

```
final DatePicker date = (DatePicker) findViewById(R.id.DatePicker01);
date.init(2011, 9, 26,
    new DatePicker.OnDateChangedListener() {
        public void onDateChanged(DatePicker view, int year,
            int monthOfYear, int dayOfMonth) {
            Date dt = new Date(year-1900,
                monthOfYear, dayOfMonth, time.getCurrentHour(),
                time.getCurrentMinute());
            text.setText(dt.toString());
        }
    });
});
```

The preceding code sets the `DatePicker.OnDateChangedListener` via a call to the `DatePicker.init()` method. The `DatePicker` control is initialized to a specific date (note that the months field is zero based, so October is month number 9, not 10). In our example, a `TextView` control is set with the date value that the user entered into the `DatePicker` control. Incidentally, the value 1900 is subtracted from the `year` parameter to make the format compatible with the `java.util.Date` class.



### Tip

Want to initialize your `DatePicker` control to the current date? You can use the default constructor of the `java.util.Date` class to determine the current date of the device.

A `TimePicker` control (also shown in Figure 8.8, bottom) is similar to the `DatePicker` control. It also doesn't have any unique attributes. However, to register for a method call when the values change, you call the more traditional method of `TimePicker`.

`setOnTimeChangedListener()`, as shown here:

```
time.setOnTimeChangedListener(new TimePicker.OnTimeChangedListener() {
    public void onTimeChanged(TimePicker view,
        int hourOfDay, int minute) {
        Date dt = new Date(date.getYear()-1900, date.getMonth(),
            date.getDayOfMonth(), hourOfDay, minute);
        text.setText(dt.toString());
    }
});
```

As in the previous example, this code also sets a `TextView` to a string displaying the time value that the user entered. When you use the `DatePicker` control and the `TimePicker` control together, the user can set both a date and a time.

## Using Indicators to Display Data to Users

The Android SDK provides a number of controls that can be used to visually show some form of information to the user. These indicator controls include progress bars, clocks, and other similar controls.

## Indicating Progress with ProgressBar

Applications commonly perform actions that can take a while. A good practice during this time is to show the user some sort of progress indicator that informs the user that the application is off “doing something.” Applications can also show how far a user is through some operation, such as a playing a song or watching a video. The Android SDK provides several types of progress bars.

The standard progress bar is a circular indicator that only animates. It does not show how complete an action is. It can, however, show that something is taking place. This is useful when an action is indeterminate in length. There are three sizes of this type of progress indicator (see Figure 8.9).

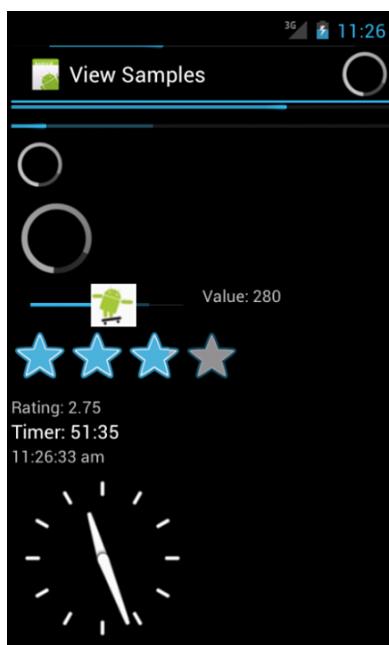


Figure 8.9 Various types of progress and rating indicators.

The second type is a horizontal progress bar that shows the completeness of an action. (For example, you can see how much of a file has downloaded.) This horizontal progress bar can also have a secondary progress indicator on it. This can be used, for instance, to show the completion of a downloading media file while that file plays.

Here is an XML layout resource definition for a basic indeterminate progress bar:

```
<ProgressBar  
    android:id="@+id/progress_bar"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content" />
```

The default style is for a medium-size circular progress indicator—not a “bar” at all. The other two styles for indeterminate progress bar are `progressBarStyleLarge` and `progressBarStyleSmall`. This style animates automatically. The next example shows the layout definition for a horizontal progress indicator:

```
<ProgressBar  
    android:id="@+id/progress_bar"  
    style="?android:attr/progressBarStyleHorizontal"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:max="100" />
```

We have also set the attribute for `max` in this example to 100. This can help mimic a percentage progress bar. That is, setting the progress to 75 shows the indicator at 75% complete.

We can set the indicator progress status programmatically as follows:

```
mProgress = (ProgressBar) findViewById(R.id.progress_bar);  
mProgress.setProgress(75);
```

You can also put these progress bars in your application’s title bar (as shown in Figure 8.9). This can save screen real estate, and can also make it easy to turn on and off an indeterminate progress indicator without changing the look of the screen. Indeterminate progress indicators are commonly used to display progress on pages where items need to be loaded before the page can finish drawing. This is often employed on web browser screens. The following code demonstrates how to place this type of indeterminate progress indicator on your `Activity` screen:

```
requestWindowFeature(Window.FEATURE_INDETERMINATE_PROGRESS);  
requestWindowFeature(Window.FEATURE_PROGRESS);  
setContentView(R.layout.indicators);  
setProgressBarIndeterminateVisibility(true);  
setProgressBarVisibility(true);  
setProgress(5000);
```

To use the indeterminate indicator on your `Activity` object’s title bar, you need to request the feature `Window.FEATURE_INDETERMINATE_PROGRESS`, as previously shown. This shows a small circular indicator in the right side of the title bar. For a horizontal progress bar style that shows behind the title, you need to enable `Window.FEATURE_PROGRESS`. These features must be enabled before your application calls the `setContentView()` method, as shown in the preceding example.

You need to know about a couple of important default behaviors. First, the indicators are visible by default. Calling the visibility methods shown in the preceding example can set their visibility on or off. Second, the horizontal progress bar defaults to a maximum progress value of 10000. In the preceding example, we set it to 5000, which is equivalent to 50%. When the value reaches the maximum value, the indicators fade away so that they aren’t visible. This happens for both indicators.

## Adjusting Progress with SeekBar

You have seen how to display progress to the user. What if, however, you want to give the user some ability to move the indicator—for example, to set the current cursor position in a playing media file or to tweak a volume setting? You accomplish this by using the `SeekBar` control provided by the Android SDK. It's like the regular horizontal progress bar, but includes a thumb, or selector, that can be dragged by the user. A default thumb selector is provided, but you can use any `drawable` item as a thumb. In Figure 8.9 (center), we replaced the default thumb with a little Android graphic.

Here we have an example of an XML layout resource definition for a simple `SeekBar`:

```
<SeekBar  
    android:id="@+id/seekbar1"  
    android:layout_height="wrap_content"  
    android:layout_width="240px"  
    android:max="500" />
```

With this sample `SeekBar`, the user can drag the thumb to any value between 0 and 500. Although this is shown visually, it might be useful to show the user what exact value the user is selecting. To do this, you can provide an implementation of the `onProgressChanged()` method, as shown here:

```
SeekBar seek = (SeekBar) findViewById(R.id.seekbar1);  
seek.setOnSeekBarChangeListener(  
    new SeekBar.OnSeekBarChangeListener() {  
        public void onProgressChanged(  
            SeekBar seekBar, int progress,boolean fromTouch) {  
            ((TextView)findViewById(R.id.seek_text))  
                .setText("Value: "+progress);  
            seekBar.setSecondaryProgress(  
                (progress+seekBar.getMax())/2);  
        }  
    });
```

There are two interesting things to notice in this example. The first is that the `fromTouch` parameter tells the code if the change came from the user input or if, instead, it came from a programmatic change as demonstrated with the regular `ProgressBar` controls. The second interesting thing is that the `SeekBar` still enables you to set a secondary progress value. In this example, we set the secondary indicator to be halfway between the user's selected value and the maximum value of the progress bar. You might use this feature to show the progress of a video and the buffer stream.

## Displaying Rating Data with RatingBar

Although the `SeekBar` is useful for allowing a user to set a value, such as the volume, the `RatingBar` has a more specific purpose: showing ratings or getting a rating from a

user. By default, this progress bar uses the star paradigm, with five stars by default. A user can drag across this horizontal to set a rating. A program can set the value, as well. However, the secondary indicator cannot be used because it is used internally by this particular control.

Here's an example of an XML layout resource definition for a `RatingBar` with four stars:

```
<RatingBar  
    android:id="@+id/ratebar1"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:numStars="4"  
    android:stepSize="0.25" />
```

This layout definition for a `RatingBar` demonstrates setting both the number of stars and the increment between each rating value. Figure 8.9 (center) illustrates how the `RatingBar` behaves. In this layout definition, a user can choose any rating value between 0 and 4.0, in increments of 0.25, the `stepSize` value. For instance, users could set a value of 2.25. This is visualized to the users, by default, with the stars partially filled.

Although the value is indicated to the user visually, you might still want to show a numeric representation of this value to the user. You can do this by implementing the `onRatingChanged()` method of the `RatingBar.OnRatingBarChangeListener` class, as shown here:

```
RatingBar rate = (RatingBar) findViewById(R.id.ratebar1);  
rate.setOnRatingBarChangeListener(new  
    RatingBar.OnRatingBarChangeListener() {  
        public void onRatingChanged(RatingBar ratingBar,  
            float rating, boolean fromTouch) {  
            ((TextView) findViewById(R.id.rating_text))  
                .setText("Rating: " + rating);  
        }  
    });
```

The preceding example shows how to register the listener. When the user selects a rating using the control, a `TextView` is set to the numeric rating the user entered. One interesting thing to note is that, unlike the `SeekBar`, the implementation of the `onRatingChange()` method is called after the change is complete, usually when the user lifts a finger. That is, while the user is dragging across the stars to make a rating, this method isn't called. It is called when the user stops pressing the control.

## Showing Time Passage with the Chronometer

Sometimes you want to show time passing instead of incremental progress. In this case, you can use the `Chronometer` control as a timer (see Figure 8.9, near the bottom). This might be useful if it's the user who is taking time doing some task or in a game where

some action needs to be timed. The `Chronometer` control can be formatted with text, as shown in this XML layout resource definition:

```
<Chronometer  
    android:id="@+id/Chronometer01"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:format="Timer: %s" />
```

You can use the `Chronometer` object's `format` attribute to put text around the time that displays. A `Chronometer` won't show the passage of time until its `start()` method is called. To stop it, simply call its `stop()` method. Finally, you can change the time from which the timer is counting. That is, you can set it to count from a particular time in the past instead of from the time it's started. You call the `setBase()` method to do this.



### Tip

The `Chronometer` uses the `elapsedRealtime()` method's time base. Passing `android.os.SystemClock.elapsedRealtime()` in to the `setBase()` method starts the `Chronometer` control at 0.

In this next code example, the timer is retrieved from the `View` by its resource identifier. We then check its base value and reset it to 0. Finally, we start the timer counting up from there.

```
final Chronometer timer =  
    (Chronometer) findViewById(R.id.Chronometer01);  
long base = timer.getBase();  
Log.d(ViewsMenu.debugTag, "base = " + base);  
timer.setBase(0);  
timer.start();
```



### Tip

You can listen for changes to the `Chronometer` by implementing the `Chronometer.OnChronometerTickListener` interface.

## Displaying the Time

Displaying the time in an application is often not necessary because Android devices have a status bar to display the current time. However, two clock controls are available to display this information: the `DigitalClock` and `AnalogClock` controls.

### Using the `DigitalClock`

The `DigitalClock` control (Figure 8.9, bottom) is a compact text display of the current time in standard numeric format based on the users' settings. It is a `TextView`, so

anything you can do with a `TextView` you can do with this control, except change its text. You can change the color and style of the text, for example.

By default, the `DigitalClock` control shows the seconds and automatically updates as each second ticks by. Here is an example of an XML layout resource definition for a `DigitalClock` control:

```
<DigitalClock  
    android:id="@+id/DigitalClock01"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content" />
```

## Using the `AnalogClock`

The `AnalogClock` control (Figure 8.9, bottom) is a dial-based clock with a basic clock face with two hands. It updates automatically as each minute passes. The image of the clock scales appropriately with the size of its `View`.

Here is an example of an XML layout resource definition for an `AnalogClock` control:

```
<AnalogClock  
    android:id="@+id/AnalogClock01"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content" />
```

The `AnalogClock` control's clock face is simple. However, you can set its minute and hour hands. You can also set the clock face to specific drawable resources, if you want to jazz it up. Neither of these clock controls accepts a different time or a static time to display. They can show only the current time in the current time zone of the device, so they are not particularly useful.

## Summary

The Android SDK provides many useful user interface components that developers can use to create compelling and easy-to-use applications. This chapter introduced you to many of the most useful controls and discussed how each behaves, how to style them, and how to handle events from the user.

You learned how controls can be combined to create user entry forms. Important controls for forms include `EditText`, `Spinner`, and various `Button` controls. You also learned about controls that can indicate progress or the passage of time to users. We talked about many common user interface controls in this chapter; however, there are many others. In the next chapter, you learn how to use various layout and container controls to organize a variety of controls on the screen easily and accurately.

## References and More Information

Android SDK Reference regarding the application `View` class:

<http://d.android.com/reference/android/view/View.html>

Android SDK Reference regarding the application `TextView` class:

<http://d.android.com/reference/android/widget/TextView.html>

Android SDK Reference regarding the application `EditText` class:

<http://d.android.com/reference/android/widget/EditText.html>

Android SDK Reference regarding the application `Button` class:

<http://d.android.com/reference/android/widget/Button.html>

Android SDK Reference regarding the application `CheckBox` class:

<http://d.android.com/reference/android/widget/CheckBox.html>

Android SDK Reference regarding the application `Switch` class:

<http://d.android.com/reference/android/widget/Switch.html>

Android SDK Reference regarding the application `RadioGroup` class:

<http://d.android.com/reference/android/widget/RadioGroup.html>

Android Dev Guide: “User Interface”:

<http://d.android.com/guide/topics/ui/index.html>

# Designing User Interfaces with Layouts

In this chapter, we discuss how to design user interfaces for Android applications. Here we focus on the various layout controls you can use to organize screen elements in different ways. We also cover some of the more complex View controls we call container views. These are View controls that can contain other View controls.

## Creating User Interfaces in Android

Application user interfaces can be simple or complex, involving many different screens or only a few. Layouts and user interface controls can be defined as application resources or created programmatically at runtime.

Although it's a bit confusing, the term *layout* is used for two different but related purposes in Android user interface design:

- In terms of resources, the `/res/layout` directory contains XML resource definitions often called layout resource files. These XML files provide a template for how to draw controls on the screen; layout resource files may contain any number of controls.
- The term is also used to refer to a set of `ViewGroup` classes, such as `LinearLayout`, `FrameLayout`, `TableLayout`, `RelativeLayout`, and `GridLayout`. These controls are used to organize other View controls. We talk more about these classes later in this chapter.

## Creating Layouts Using XML Resources

As discussed in previous chapters, Android provides a simple way to create layout resource files in XML. These resources are stored in the `/res/layout` project directory hierarchy. This is the most common and convenient way to build Android user interfaces and is especially useful for defining screen elements and default control properties that

you know about at compile time. These layout resources are then used much like templates. They are loaded with default attributes that you can modify programmatically at runtime.

You can configure almost any `ViewGroup` or `View` (or `View` subclass) attribute using the XML layout resource files. This method greatly simplifies the user interface design process, moving much of the static creation and layout of user interface controls, and basic definition of control attributes, to the XML instead of littering the code.

Developers reserve the ability to alter these layouts programmatically as necessary, but they can set all the defaults in the XML template.

You'll recognize the following as a simple layout file with a `LinearLayout` and a single `TextView` control. Here is the default layout file provided with any new Android project in Eclipse, referred to as `/res/layout/main.xml`:

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

This block of XML shows a basic layout with a single `TextView` control. The first line, which you might recognize from most XML files, is required with the android layout namespace, as shown. Because it's common across all the files, we do not show it in any other examples.

Next, we have the `LinearLayout` element. `LinearLayout` is a `ViewGroup` that shows each child `View` either in a single column or in a single row. When applied to a full screen, it merely means that each child `View` is drawn under the previous `View` if the orientation is set to vertical or to the right of the previous `View` if the orientation is set to horizontal.

Finally, there is a single child `View`—in this case, a `TextView`. A `TextView` is a control that is also a `View`. A `TextView` draws text on the screen. In this case, it draws the text defined in the "`@string/hello`" string resource.

Creating only an XML file, though, won't actually draw anything on the screen. A particular layout is usually associated with a particular `Activity`. In your default Android project, there is only one activity, which sets the `main.xml` layout by default. To associate the `main.xml` layout with the activity, use the method call `setContentView()` with the identifier of the `main.xml` layout. The ID of the layout matches the XML file-name without the extension. In this case, the preceding example came from `main.xml`, so the identifier of this layout is simply `main`:

```
setContentView(R.layout.main);
```



### Warning

The Eclipse layout resource designer can be a helpful tool for designing and previewing layout resources. However, the preview can't replicate exactly how the layout appears to end users. For this, you must test your application on a properly configured emulator and, more importantly, on your target devices.



### Tip

The code examples provided in this section are taken from the *SameLayout* application. The source code for the *SameLayout* application is provided for download on the book's websites.

The following example shows how to programmatically have an `Activity` instantiate a `LinearLayout` and place two `TextView` controls within it as child controls. The same two string resources are used for the contents of the controls; these actions are done at runtime instead.

```
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
  
    TextView text1 = new TextView(this);  
    text1.setText(R.string.string1);  
  
    TextView text2 = new TextView(this);  
    text2.setText(R.string.string2);  
    text2.setTextSize(TypedValue.COMPLEX_UNIT_SP, 60);  
  
    LinearLayout ll = new LinearLayout(this);  
    ll.setOrientation(LinearLayout.VERTICAL);  
    ll.addView(text1);  
    ll.addView(text2);  
  
    setContentView(ll);  
}
```

The `onCreate()` method is called when the `Activity` is created. The first thing this method does is some normal housekeeping by calling the constructor for the base class.

Next, two `TextView` controls are instantiated. The `Text` property of each `TextView` is set using the `setText()` method. All `TextView` attributes, such as `TextSize`, are set by making method calls on the `TextView` control. These actions perform the same function that you have in the past by setting the properties `Text` and `TextSize` using the Eclipse layout resource designer, except these properties are set at runtime instead of defined in the layout files compiled into your application package.



### Tip

The XML property name is usually similar to the method calls for getting and setting that same control property programmatically. For instance, `android:visibility` maps to the methods `setVisibility()` and `getVisibility()`. In the preceding sample `TextView`, the methods for getting and setting the `TextSize` property are `getTextSize()` and `setTextSize()`.

To display the `TextView` controls appropriately, we need to encapsulate them within a container of some sort (a layout). In this case, we use a `LinearLayout` with the orientation set to `VERTICAL` so that the second `TextView` begins beneath the first, each aligned to the left of the screen. The two `TextView` controls are added to the `LinearLayout` in the order we want them to display.

Finally, we call the `setContentView()` method, part of your `Activity` class, to draw the `LinearLayout` and its contents on the screen.

As you can see, the code can rapidly grow in size as you add more view controls and you need more attributes for each view. Here is that same layout, now in an XML layout file:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:id="@+id/TextView1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/string1" />
    <TextView
        android:id="@+id/TextView2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="60sp"
        android:text="@string/string2" />
</LinearLayout>
```

You might notice that this isn't a literal translation of the code example from the previous section, although the output is identical, as shown in Figure 9.1.

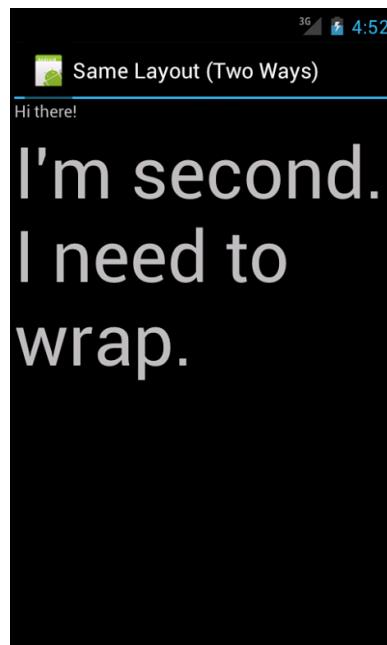


Figure 9.1 Two different methods to create a screen have the same result.

First, in the XML layout files, `layout_width` and `layout_height` are required attributes. Next, you see that each `TextView` control has a unique `id` property assigned so that it can be accessed programmatically at runtime. Finally, the `textSize` property needs to have its units defined. The XML attribute takes a `dimension` type.

The end result differs only slightly from the programmatic method. However, it's far easier to read and maintain. Now you need only one line of code to display this layout view. Again, the layout resource is stored in the `/res/layout/resource_based_layout.xml` file:

```
setContentView(R.layout.resource_based_layout);
```

## Organizing Your User Interface

In Chapter 8, “Exploring User Interface Screen Elements,” we talk about how the class `View` is the building block for user interfaces in Android. All user interface controls, such as `Button`, `Spinner`, and `EditText`, derive from the `View` class.

Now we talk about a special kind of `View` called a `ViewGroup`. The classes derived from `ViewGroup` enable developers to display `View` controls such as `TextView` and `Button` controls on the screen in an organized fashion.

It's important to understand the difference between `View` and `ViewGroup`. Like other `View` controls, including the controls from the previous chapter, `ViewGroup` controls represent a rectangle of screen space. What makes `ViewGroup` different from your typical control is that `ViewGroup` objects contain other `View` controls. A `View` that contains other `View` controls is called a *parent view*. The parent `View` contains `View` controls called *child views*, or *children*.

You add child `View` controls to a `ViewGroup` programmatically using the method `addView()`. In XML, you add child objects to a `ViewGroup` by defining the child `View` control as a child node in the XML (within the parent XML element, as we've seen various times using the `LinearLayout` `ViewGroup`).

`ViewGroup` subclasses are broken down into two categories:

- Layout classes
- View container controls

## Using `ViewGroup` Subclasses for Layout Design

Many of the most important subclasses of `ViewGroup` used for screen design end with “Layout.” For example, the most common layout classes are `LinearLayout`, `RelativeLayout`, `TableLayout`, and `FrameLayout`. You can use each of these classes to position other `View` controls on the screen in different ways. For example, we've been using the `LinearLayout` to arrange various `TextView` and `EditText` controls on the screen in a single vertical column. Users do not generally interact with the layouts directly. Instead, they interact with the `View` controls they contain.

## Using `ViewGroup` Subclasses as View Containers

The second category of `ViewGroup` subclasses is the indirect “subclasses”—some formally, and some informally. These special `View` controls act as `View` containers like `Layout` objects do, but they also provide some kind of active functionality that enables users to interact with them like other controls. Unfortunately, these classes are not known by any handy names; instead, they are named for the kind of functionality they provide.

Some of the classes that fall into this category include `Gallery`, `GridView`, `ImageSwitcher`, `ScrollView`, `TabHost`, and `ListView`. It can be helpful to consider these objects as different kinds of `View` browsers, or container classes. A `ListView` displays each `View` control as a list item, and the user can browse between the individual

controls using vertical scrolling capability. A `Gallery` is a horizontal scrolling list of `View` controls with a center “current” item; the user can browse the `View` controls in the `Gallery` by scrolling left and right. A `TabHost` is a more complex `View` container, where each tab can contain a `View` (such as a layout) and the user selects a tab to see its contents.

## Using Built-in Layout Classes

We talked a lot about the `LinearLayout` layout, but there are several other types of layouts. Each layout has a different purpose and order in which it displays its child `View` controls on the screen. Layouts are derived from `android.view.ViewGroup`.

The types of layouts built in to the Android SDK framework include the following:

- `FrameLayout`
- `LinearLayout`
- `RelativeLayout`
- `TableLayout`
- `GridLayout`

### Tip

Many of the code examples provided in this section are taken from the `SimpleLayout` application. The source code for the `SimpleLayout` application is provided for download on the book’s websites.

All layouts, regardless of their type, have basic layout attributes. Layout attributes apply to any child `View` control within that layout. You can set layout attributes at runtime programmatically, but ideally you set them in the XML layout files using the following syntax:

```
android:layout_attribute_name="value"
```

There are several layout attributes that all `ViewGroup` objects share. These include size attributes and margin attributes. You can find basic layout attributes in the `ViewGroup.LayoutParams` class. The margin attributes enable each child `View` within a layout to have padding on each side. Find these attributes in the `ViewGroup.MarginLayoutParams` classes. There are also a number of `ViewGroup` attributes for handling child `View` drawing bounds and animation settings.

Some of the important attributes shared by all `ViewGroup` subtypes are shown in Table 9.1.

Table 9.1 Important ViewGroup Attributes

Attribute Name (All begin with android:)	Applies To	Description	Value
layout_height	Parent view	Height of the view. Used on attribute for child view controls within layouts. Required in some layouts, optional in others.	Dimension value or match_parent or wrap_content
	Child view		
layout_width	Parent view	Width of the view. Used on attribute for child view controls within layouts. Required in some layouts, optional in others.	Dimension value or match_parent or wrap_content
	Child view		
layout_margin	Parent view	Extra space around all sides of view.	Dimension value. Use more specific margin attributes to control individual margin sides, if necessary.
	Child view		

Here's an XML layout resource example of a `LinearLayout` set to the size of the screen, containing one `TextView` that is set to its full height and the width of the `LinearLayout` (and therefore the screen):

```
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:id="@+id/TextView01"
        android:layout_height="match_parent"
        android:layout_width="match_parent" />
</LinearLayout>
```

Here is an example of a `Button` object with some margins set via XML used in a layout resource file:

```
<Button
    android:id="@+id/Button01"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Press Me"
```

```
    android:layout_marginRight="20px"  
    android:layout_marginTop="60px" />
```

Remember that a layout element can cover any rectangular space on the screen; it doesn't need to be the entire screen. Layouts can be nested within one another. This provides great flexibility when developers need to organize screen elements. It is common to start with a `FrameLayout` or `LinearLayout` as the parent layout for the entire screen and then organize individual screen elements inside the parent layout using whichever layout type is most appropriate.

Now let's talk about each of the common layout types individually and how they differ from one another.

## Using `FrameLayout`

A `FrameLayout` view is designed to display a stack of child `View` items. You can add multiple views to this layout, but each `View` is drawn from the top-left corner of the layout. You can use this to show multiple images within the same region, as shown in Figure 9.2, and the layout is sized to the largest child `View` in the stack.

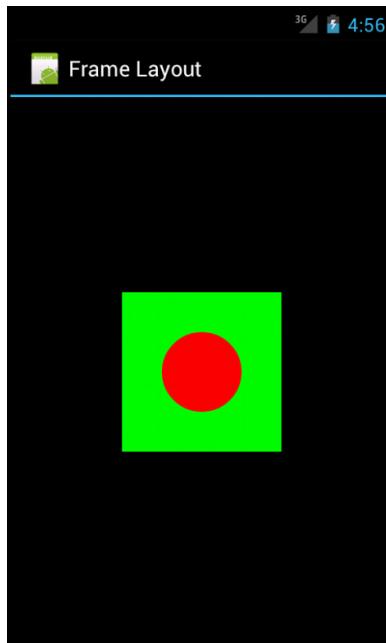


Figure 9.2 An example of `FrameLayout` usage.

You can find the layout attributes available for `FrameLayout` child view controls in `android.widget.FrameLayout.LayoutParams`. Table 9.2 describes some of the important attributes specific to `FrameLayout` views.

Table 9.2 Important `FrameLayout` View Attributes

Attribute Name (All begin with <code>android:</code> ) Applies To	Description	Value
<code>foreground</code> Parent view	Drawable to draw over the content.	Drawable resource.
<code>foregroundGravity</code> Parent view	Gravity of foreground drawable.	One or more constants separated by “ ”. The constants available are <code>top</code> , <code>bottom</code> , <code>left</code> , <code>right</code> , <code>center_vertical</code> , <code>fill_vertical</code> , <code>center_horizontal</code> , <code>fill_horizontal</code> , <code>center</code> , and <code>fill</code> .
<code>measureAllChildren</code> Parent view	Restrict size of layout to all child views or just the child views set to <code>VISIBLE</code> (and not those set to <code>INVISIBLE</code> ).	True or false.
<code>layout_gravity</code> Child view	A gravity constant that describes how to place the child view within the parent.	One or more constants separated by “ ”. The constants available are <code>top</code> , <code>bottom</code> , <code>left</code> , <code>right</code> , <code>center_vertical</code> , <code>fill_vertical</code> , <code>center_horizontal</code> , <code>fill_horizontal</code> , <code>center</code> , and <code>fill</code> .

Here's an example of an XML layout resource with a `FrameLayout` and two child `View` controls, both `ImageView` controls. The green rectangle is drawn first and the red oval is drawn on top of it. The green rectangle is larger, so it defines the bounds of the `FrameLayout`:

```
<FrameLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:id="@+id/FrameLayout01"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center">
    <ImageView
        android:id="@+id/ImageView01"
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/green_rect"
    android:minHeight="200px"
    android:minWidth="200px" />
<ImageView
    android:id="@+id/ImageView02"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/red_oval"
    android:minHeight="100px"
    android:minWidth="100px"
    android:layout_gravity="center" />
</FrameLayout>
```

## Using LinearLayout

A `LinearLayout` view organizes its child view controls in a single row, as shown in Figure 9.3, or a single column, depending on whether its `orientation` attribute is set to horizontal or vertical. This is a very handy layout method for creating forms.

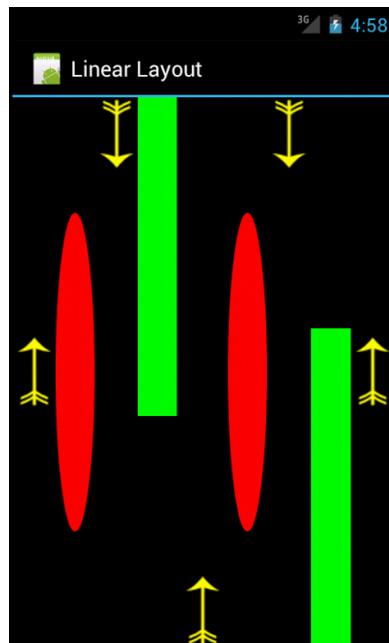


Figure 9.3 An example of `LinearLayout` (horizontal orientation).

You can find the layout attributes available for `LinearLayout` child View controls in `android.widget.LinearLayout.LayoutParams`. Table 9.3 describes some of the important attributes specific to `LinearLayout` views.

Table 9.3 Important `LinearLayout` View Attributes

Attribute Name (All begin with <code>android:</code> )	Applies To	Description	Value
<code>orientation</code>	Parent view	Layout is a single row (horizontal) or single column (vertical) of controls.	Either horizontal or vertical.
<code>gravity</code>	Parent view	Gravity of child views within layout.	One or more constants separated by " ". The constants available are <code>top</code> , <code>bottom</code> , <code>left</code> , <code>right</code> , <code>center_vertical</code> , <code>fill_vertical</code> , <code>center_horizontal</code> , <code>fill_horizontal</code> , <code>center</code> , and <code>fill</code> .
<code>weightSum</code>	Parent view	Sum of all child control weights.	A number that defines the sum of all child control weights. Default is 1.
<code>layout_gravity</code>	Child view	The gravity for a specific child view. Used for positioning of views.	One or more constants separated by " ". The constants available are <code>top</code> , <code>bottom</code> , <code>left</code> , <code>right</code> , <code>center_vertical</code> , <code>fill_vertical</code> , <code>center_horizontal</code> , <code>fill_horizontal</code> , <code>center</code> , and <code>fill</code> .
<code>layout_weight</code>	Child view	The weight for a specific child view. Used to provide ratio of screen space used within the parent control.	The sum of values across all child views in a parent view must equal the <code>weightSum</code> attribute of the parent <code>LinearLayout</code> control. For example, one child control might have a value of .3 and another a value of .7.

## Using RelativeLayout

The `RelativeLayout` view enables you to specify where the child view controls are in relation to each other. For instance, you can set a child view to be positioned “above” or “below” or “to the left of” or “to the right of” another View, referred to by its unique identifier. You can also align child View controls relative to one another or the parent layout edges. Combining `RelativeLayout` attributes can simplify creating interesting user interfaces without resorting to multiple layout groups to achieve a desired effect. Figure 9.4 shows how the button controls are relative to each other.

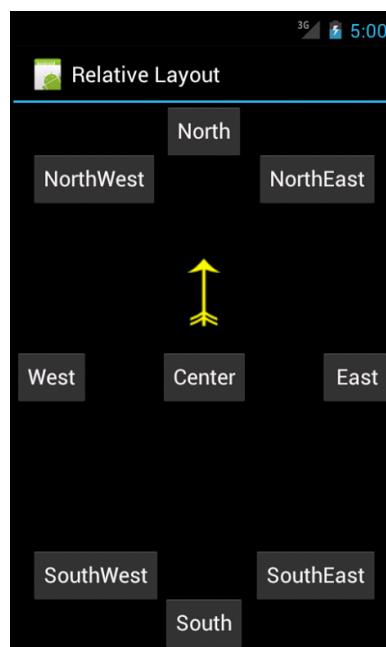


Figure 9.4 An example of `RelativeLayout` usage.

You can find the layout attributes available for `RelativeLayout` child View controls in `android.widget.RelativeLayout.LayoutParams`. Table 9.4 describes some of the important attributes specific to `RelativeLayout` views.

Table 9.4 Important `RelativeLayout` View Attributes

Attribute Name (All begin with <code>android:</code> )	Applies To	Description	Value
<code>gravity</code>	Parent view	Gravity of child views within layout.	One or more constants separated by " ". The constants available are <code>top</code> , <code>bottom</code> , <code>left</code> , <code>right</code> , <code>center</code> , <code>vertical</code> , <code>fill</code> , <code>vertical_center</code> , <code>horizontal</code> , <code>fill_center</code> , <code>horizontal_center</code> , and <code>fill</code> .
<code>layout_centerInParent</code>	Child view	Centers child view horizontally and vertically within parent view.	True or false.
<code>layout_centerHorizontal</code>	Child view	Centers child view horizontally within parent view.	True or false.
<code>layout_centerVertical</code>	Child view	Centers child view vertically within parent view.	True or false.
<code>layout_alignParentTop</code>	Child view	Aligns child view with top edge of parent view.	True or false.
<code>layout_alignParentBottom</code>	Child view	Aligns child view with bottom edge of parent view.	True or false.
<code>layout_alignParentLeft</code>	Child view	Aligns child view with left edge of parent view.	True or false.
<code>layout_alignParentRight</code>	Child view	Aligns child view with right edge of parent view.	True or false.
<code>layout_alignRight</code>	Child view	Aligns right edge of child view with right edge of another child view, specified by ID.	A view ID; for example, <code>@+id/Button1</code>
<code>layout_alignLeft</code>	Child view	Aligns left edge of child view with left edge of another child view, specified by ID.	A view ID; for example, <code>@+id/Button1</code>

**Attribute Name**

<b>(All begin with android:)</b>	<b>Applies To</b>	<b>Description</b>	<b>Value</b>
layout_alignTop	Child view	Aligns top edge of child view with top edge of another child view, specified by ID.	A view ID; for example, @+id/Button1
layout_alignBottom	Child view	Aligns bottom edge of child view with bottom edge of another child view, specified by ID.	A view ID; for example, @+id/Button1
layout_above	Child view	Positions bottom edge of child view above another child view, specified by ID.	A view ID; for example, @+id/Button1
layout_below	Child view	Positions top edge of child view below another child view, specified by ID.	A view ID; for example, @+id/Button1
layout_toLeftOf	Child view	Positions right edge of child view to the left of another child view, specified by ID.	A view ID; for example, @+id/Button1
layout_toRightOf	Child view	Positions left edge of child view to the right of another child view, specified by ID.	A view ID; for example, @+id/Button1

Here's an example of an XML layout resource with a `RelativeLayout` and two child view controls—a `Button` object aligned relative to its parent, and an `ImageView` aligned and positioned relative to the `Button` (and the parent):

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:id="@+id/RelativeLayout01"
    android:layout_height="match_parent"
    android:layout_width="match_parent">
    <Button
        android:id="@+id/ButtonCenter"
        android:text="Center"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true" />
```

```
<ImageView  
    android:id="@+id/ImageView01"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_above="@+id/ButtonCenter"  
    android:layout_centerHorizontal="true"  
    android:src="@drawable/arrow" />  
</RelativeLayout>
```



### Warning

The `AbsoluteLayout` class has been deprecated. `AbsoluteLayout` uses specific x and y coordinates for child view placement. This layout can be useful when pixel-perfect placement is required. However, it's less flexible because it does not adapt well to other device configurations with different screen sizes and resolutions. Under most circumstances, other popular layout types such as `FrameLayout` and `RelativeLayout` suffice in place of `AbsoluteLayout`, so we encourage you to use these layouts instead when possible.

## Using TableLayout

A `TableLayout` view organizes children into rows, as shown in Figure 9.5. You add individual `View` controls within each row of the table using a `TableRow` layout `View` (which is basically a horizontally oriented `LinearLayout`) for each row of the table. Each column of the `TableRow` can contain one `View` (or layout with child `View` controls). You place `View` items added to a `TableRow` in columns in the order they are added. You can specify the column number (zero based) to skip columns as necessary (the bottom row shown in Figure 9.5 demonstrates this); otherwise, the `View` control is put in the next column to the right. Columns scale to the size of the largest `View` of that column. You can also include normal `View` controls instead of `TableRow` elements, if you want the `View` to take up an entire row.

You can find the layout attributes available for `TableLayout` child `View` controls in `android.widget.TableLayout.LayoutParams`. You can find the layout attributes available for `TableRow` child `View` controls in `android.widget.TableRow.LayoutParams`. Table 9.5 describes some of the important attributes specific to `TableLayout` controls.

Table 9.5 Important `TableLayout` and `TableRow` View Attributes

#### Attribute Name

(All begin with `android:`) Applies To

Description

Value

<code>collapseColumns</code>	<code>TableLayout</code>	A comma-delimited list of column indices to collapse (zero-based)	String or string resource. For example, "0,1,3,5".
------------------------------	--------------------------	---	--

**Attribute Name****(All begin with android:)** **Applies To**

		Description	Value
shrinkColumns	TableLayout	A comma-delimited list of column indices to shrink (zero-based)	String or string resource. Use "*" for all columns. For example, "0,1,3,5".
stretchColumns	TableLayout	A comma-delimited list of column indices to stretch (zero-based)	String or string resource. Use "*" for all columns. For example, "0,1,3,5".
layout_column	TableRow child view	Index of column this child view should be displayed in (zero-based)	Integer or integer resource. For example, 1.
layout_span	TableRow child view	Number of columns this child view should span across	Integer or integer resource greater than or equal to 1. For example, 3.

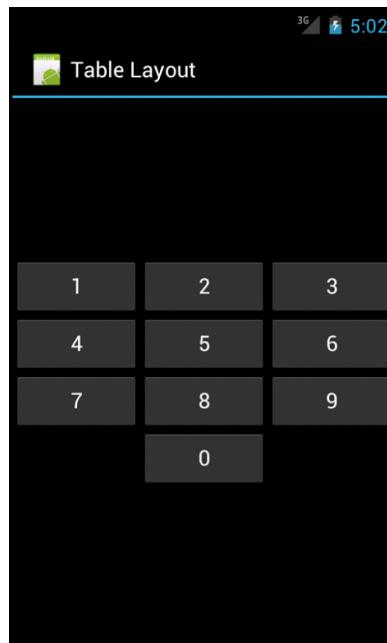


Figure 9.5 An example of TableLayout usage.

Here's an example of an XML layout resource with a `TableLayout` with two rows (two `TableRow` child objects). The `TableLayout` is set to stretch the columns to the size of the screen width. The first `TableRow` has three columns; each cell has a `Button` object. The second `TableRow` puts only one `Button` control into the second column explicitly:

```
<TableLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:id="@+id/TableLayout01"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:stretchColumns="*">
    <TableRow
        android:id="@+id/TableRow01">
        <Button
            android:id="@+id/ButtonLeft"
            android:text="Left Door" />
        <Button
            android:id="@+id/ButtonMiddle"
            android:text="Middle Door" />
        <Button
            android:id="@+id/ButtonRight"
            android:text="Right Door" />
    </TableRow>
    <TableRow
        android:id="@+id/TableRow02">
        <Button
            android:id="@+id/ButtonBack"
            android:text="Go Back"
            android:layout_column="1" />
    </TableRow>
</TableLayout>
```

## Using GridLayout

Introduced in Android 4.0 (API Level 14), the `GridLayout` organizes its children inside a grid. But don't confuse it with `GridView`; this layout grid is dynamically created. Unlike a `TableLayout`, child view controls in a `GridLayout` can span rows and columns and are more flat and efficient in terms of layout rendering. In fact, it is the child view controls of a `GridLayout` that tell the layout where they want to be placed. Figure 9.6 shows an example of a `GridLayout` with five child controls.

You can find the layout attributes available for `GridLayout` child view controls in `android.widget.GridLayout.LayoutParams`. Table 9.6 describes some of the important attributes specific to `GridLayout` controls.

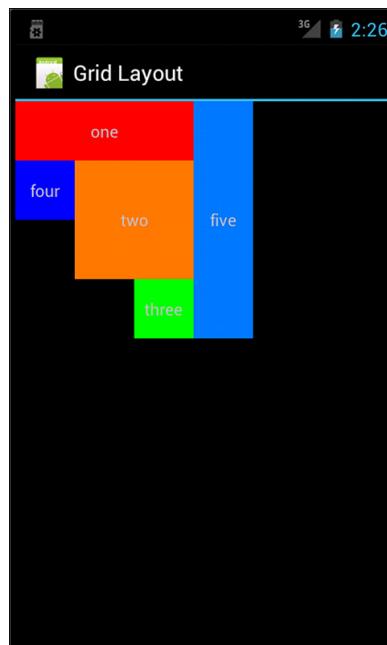


Figure 9.6 An example of GridLayout usage.

Table 9.6 Important GridLayout View Attributes

Attribute Name (All begin with android:)	Applies To	Description	Value
columnCount	GridLayout	Defines a fixed number of columns for the grid.	A whole number; for example "4".
rowCount	GridLayout	Defines a fixed number of rows for the grid.	A whole number; for example "3".
orientation	GridLayout	When a row or column value is not specified on a child, this is used to determine whether the next child is down a row or over a column.	Can be vertical (down a row) or horizontal (over a column).

Table 9.6 **Continued**

<b>Attribute Name</b>	<b>(All begin with <code>android:</code>) Applies To</b>	<b>Description</b>	<b>Value</b>
<code>layout_column</code>	Child view of <code>GridLayout</code>	Index of column this child view should be displayed in (zero-based).	Integer or integer resource. For example, 1.
<code>layout_columnSpan</code>	Child view of <code>GridLayout</code>	Number of columns this child view should span across.	Integer or integer resource greater than or equal to 1. For example, 3
<code>layout_row</code>	Child view of <code>GridLayout</code>	Index of row this child view should be displayed in (zero-based).	Integer or integer resource. For example, 1.
<code>layout_rowSpan</code>	Child view of <code>GridLayout</code>	Number of columns this child view should span down.	Integer or integer resource greater than or equal to 1. For example, 3.
<code>layout_gravity</code>	Child view of <code>GridLayout</code>	Specifies the “direction” in which the view should be placed within the grid cells it will occupy.	One or more constants separated by “ ”. Some of the constants available are <code>baseline</code> , <code>top</code> , <code>bottom</code> , <code>left</code> , <code>right</code> , <code>center_vertical</code> , <code>fill_vertical</code> , <code>center_horizontal</code> , <code>fill_horizontal</code> , <code>center</code> , and <code>fill</code> . Defaults to <code>baseline left</code> .

The following is an example of an XML layout resource with a `GridLayout` view resulting in four rows and four columns. Each child control occupies a certain number of rows and columns. Because the default span attribute value is 1, we only specify when the element will take up more than one row or column. For instance, the first `TextView` is one row high and three columns wide. The height and width of each of the View controls is specified to control the look of the result; otherwise, the `GridLayout` control will automatically assign sizing.

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/gridLayout1"
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent"
    android:columnCount="4"
    android:rowCount="4" >

<TextView
    android:layout_width="150dp"
    android:layout_height="50dp"
    android:layout_column="0"
    android:layout_columnSpan="3"
    android:layout_row="0"
    android:background="#ff0000"
    android:gravity="center"
    android:text="one" />

<TextView
    android:layout_width="100dp"
    android:layout_height="100dp"
    android:layout_column="1"
    android:layout_columnSpan="2"
    android:layout_row="1"
    android:layout_rowSpan="2"
    android:background="#ff7700"
    android:gravity="center"
    android:text="two" />

<TextView
    android:layout_width="50dp"
    android:layout_height="50dp"
    android:layout_column="2"
    android:layout_row="3"
    android:background="#00ff00"
    android:gravity="center"
    android:text="three" />

<TextView
    android:layout_width="50dp"
    android:layout_height="50dp"
    android:layout_column="0"
    android:layout_row="1"
    android:background="#0000ff"
    android:gravity="center"
    android:text="four" />

<TextView
```

```
    android:layout_width="50dp"
    android:layout_height="200dp"
    android:layout_column="3"
    android:layout_row="0"
    android:layout_rowSpan="4"
    android:background="#0077ff"
    android:gravity="center"
    android:text="five" />

</GridLayout>
```

## Using Multiple Layouts on a Screen

Combining different layout methods on a single screen can create complex layouts. Remember that because a layout contains View controls and is, itself, a View control, it can contain other layouts.

### Tip



Want to create a certain amount of space between View controls without using a nested layout? Check out the new Space view (`android.widget.Space`) added in Ice Cream Sandwich (Android 4.0).

Figure 9.7 demonstrates a combination of layout views used in conjunction to create a more complex and interesting screen.

### Warning



Keep in mind that individual screens of mobile applications should remain sleek and relatively simple. This is not just because this design results in a more positive user experience; cluttering your screens with complex (and deep) View hierarchies can lead to performance problems. Use the Hierarchy Viewer to inspect your application layouts; you can also use the `layoutopt` command-line tool to help optimize your layouts and identify unnecessary components. This tool often helps identify opportunities to use layout optimization techniques, such as the `<merge>` and `<include>` tags.

## Using Container Control Classes

Layouts are not the only controls that can contain other View controls. Although layouts are useful for positioning other View controls on the screen, they aren't interactive. Now let's talk about the other kind of ViewGroup: the containers. These View controls encapsulate other, simpler View types and give the user the ability to interactively browse the child View controls in a standard fashion. Much like layouts, these controls each have a special, well-defined purpose.

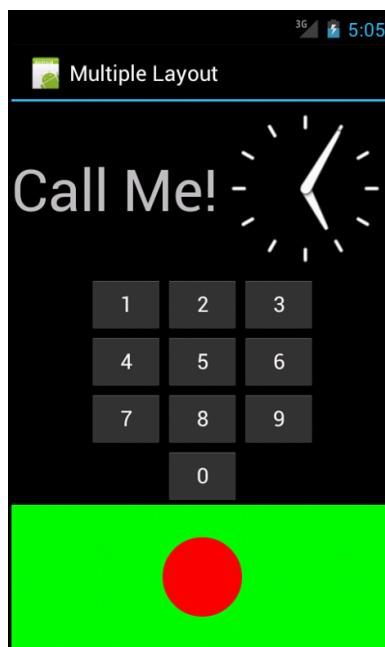


Figure 9.7 An example of multiple layouts used together.

The types of `ViewGroup` containers built in to the Android SDK framework include

- Lists, grids, and galleries
- Tabs with `TabHost` and `TabControl`
- `ScrollView` and `HorizontalScrollView` for scrolling
- `ViewFlipper`, `ImageSwitcher`, and `TextSwitcher` for switching
- `SlidingDrawer` for hiding and showing content

#### Tip

Many of the code examples provided in this chapter are taken from the `AdvancedLayouts` application. The source code for the `AdvancedLayouts` application is provided for download on the book's websites.

## Using Data-Driven Containers

Some of the `View` container controls are designed for displaying repetitive `View` controls in a particular way. Examples of this type of `View` container control include `ListView`, `GridView`, and `GalleryView`:

- **ListView:** Contains a vertically scrolling, horizontally filled list of view controls, each of which typically contains a row of data; the user can choose an item to perform some action upon.
- **GridView:** Contains a grid of view controls, with a specific number of columns; this container is often used with image icons; the user can choose an item to perform some action upon.
- **GalleryView:** Contains a horizontally scrolling list of view controls, also often used with image icons; the user can select an item to perform some action upon.

These containers are all types of AdapterView controls. An AdapterView control contains a set of child view controls to display data from some data source. An Adapter generates these child view controls from a data source. Because this is an important part of all these container controls, we talk about the Adapter objects first.

In this section, you learn how to bind data to view controls using Adapter objects. In the Android SDK, an Adapter reads data from some data source and generates the data for a View control based on some rules, depending on the type of Adapter used. This View is used to populate the child View controls of a particular AdapterView.

The most common Adapter classes are the CursorAdapter and the ArrayAdapter. The CursorAdapter gathers data from a Cursor, whereas the ArrayAdapter gathers data from an array. A CursorAdapter is a good choice to use when using data from a database. The ArrayAdapter is a good choice to use when there is only a single column of data or when the data comes from a resource array.

You should know some common elements of Adapter objects. When creating an Adapter, you provide a layout identifier. This layout is the template for filling in each row of data. The template you create contains identifiers for particular controls that the Adapter assigns data to. A simple layout can contain as little as a single TextView control. When making an Adapter, refer to both the layout resource and the identifier of the TextView control. The Android SDK provides some common layout resources for use in your application.

## Using ArrayAdapter

An ArrayAdapter binds each element of the array to a single View control within the layout resource. Here is an example of creating an ArrayAdapter:

```
private String[] items = {  
    "Item 1", "Item 2", "Item 3" };  
ArrayAdapter adapt =  
    new ArrayAdapter<String>  
        (this, R.layout.textview, items);
```

In this example, we have a String array called `items`. This is the array used by the ArrayAdapter as the source data. We also use a layout resource, which is the View that is repeated for each item in the array. This is defined as follows:

```
<TextView xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textSize="20px" />
```

This layout resource contains only a single `TextView`. However, you can use a more complex layout with constructors that also take the resource identifier of a `TextView` within the layout. Each child view within the `AdapterView` that uses this `Adapter` gets one `TextView` instance with one of the strings from the `String` array.

If you have an array resource defined, you can also directly set the `entries` attribute for an `AdapterView` to the resource identifier of the array to automatically provide the  `ArrayAdapter`.

## Using CursorAdapter

A `CursorAdapter` binds one or more columns of data to one or more `View` controls within the layout resource provided. This is best shown with an example. We will also discuss `Cursor` objects later in this book and in *Android Wireless Application Development, Volume II: Advanced Topics* as well, when we discuss databases and content providers.

The following example demonstrates creating a `CursorAdapter` by querying the `Contacts` content provider. The `CursorAdapter` requires the use of a `Cursor`.

```
Cursor names = managedQuery(
    Contacts.Phones.CONTENT_URI, null, null, null, null);
startManagingCursor(names);
ListAdapter adapter = new SimpleCursorAdapter(
    this, R.layout.two_text,
    names, new String[] {
        Contacts.Phones.NAME,
        Contacts.Phones.NUMBER
    }, new int[] {
        R.id.scratch_text1,
        R.id.scratch_text2
});
```

In this example, we present a couple of new concepts. First, you need to know that the `Cursor` must contain a field named `_id`. In this case, we know that the `Contacts` content provider does have this field. This field is used later when we handle the user selecting a particular item.

### Note

Although the `Contacts` class has been deprecated by the `ContactsContract` class introduced in Android 2.0, `Contacts` is the only method for accessing contact information that works on both older and newer editions of Android without modification. We talk more about `Contacts` and content providers in Chapter 14, “Using Content Providers.”

We make a call to `managedQuery()` to get the Cursor. Then, we instantiate a `SimpleCursorAdapter` as a `ListAdapter`. Our layout, `R.layout.two_text`, has two `TextView` controls in it, which are used in the last parameter. `SimpleCursorAdapter` enables us to match up columns in the database with particular controls in our layout. For each row returned from the query, we get one instance of the layout within our `AdapterView`.

### Binding Data to the AdapterView

Now that you have an `Adapter` object, you can apply this to one of the `AdapterView` controls. Any of them works. Although the `Gallery` technically takes a `SpinnerAdapter`, the instantiation of `SimpleCursorAdapter` also returns a `SpinnerAdapter`. Here is an example of this with a `ListView`, continuing on from the previous sample code:

```
((ListView) findViewById(R.id.list)).setAdapter(adapter);
```

The call to the `setAdapter()` method of the `AdapterView`, a `ListView` in this case, should come after your call to `setContentView()`. This is all that is required to bind data to your `AdapterView`. Figure 9.8 shows the same data in a `ListView`, `Gallery`, and `GridView`.

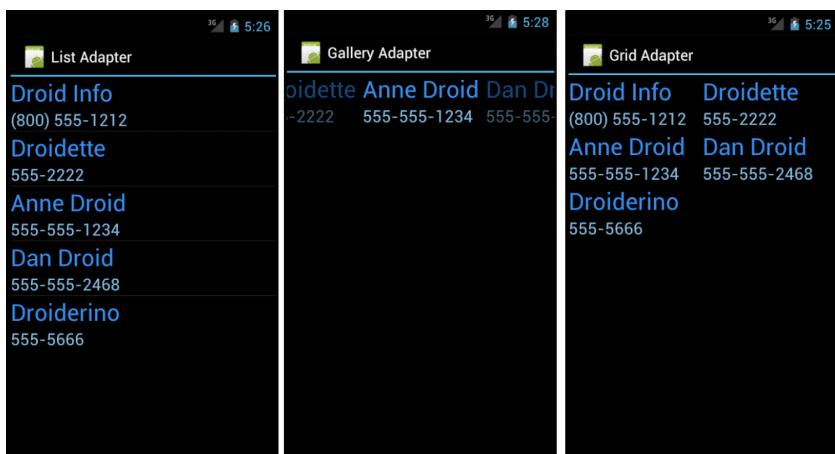


Figure 9.8 `ListView`, `Gallery`, and `GridView`: same data, same list item, different layout views.

### Handling Selection Events

You often use `AdapterView` controls to present data from which the user should select. All three of the discussed controls—`ListView`, `GridView`, and `Gallery`—enable your application to monitor for click events in the same way. You need to call `setOnItemClickListener()` on your `AdapterView` and pass in an implementation of

the `AdapterView.OnItemClickListener` class. Here is a sample implementation of this class:

```
av.setOnItemClickListener(
    new AdapterView.OnItemClickListener() {
        public void onItemClick(
            AdapterView<?> parent, View view,
            int position, long id) {
            Toast.makeText(Scratch.this, "Clicked _id="+id,
                Toast.LENGTH_SHORT).show();
        }
});
```

In the preceding example, `av` is our `AdapterView`. The implementation of the `onItemClick()` method is where all the interesting work happens. The `parent` parameter is the `AdapterView` where the item was clicked. This is useful if your screen has more than one `AdapterView` on it. The `View` parameter is the specific `View` within the item that was clicked. The `position` is the zero-based position within the list of items that the user selects. Finally, the `id` parameter is the value of the `_id` column for the particular item that the user selects. This is useful for querying for further information about that particular row of data that the item represents.

Your application can also listen for long-click events on particular items. Additionally, your application can listen for selected items. Although the parameters are the same, your application receives a call as the highlighted item changes. This can be in response to the user scrolling with the arrow keys and not selecting an item for action.

## Using ListActivity

The `ListView` control is commonly used for full-screen menus or lists of items from which a user selects. As such, you might consider using `ListActivity` as the base class for such screens. Using the `ListActivity` can simplify these types of screens.

First, to handle item events, you now need to provide an implementation in your `ListActivity`. For instance, the equivalent of `onListItemClickListener` is to implement the `onListItemClick()` method within your `ListActivity`.

Second, to assign an `Adapter`, you need a call to the `setListAdapter()` method. You do this after the call to the `setContentView()` method. However, this hints at some of the limitations of using `ListActivity`.

To use `ListActivity`, the layout that is set with the `setContentView()` method must contain a `ListView` with the identifier set to `android:list`; this cannot be changed. Second, you can also have a `View` with an identifier set to `android:empty` to have a `View` display when no data is returned from the `Adapter`. Finally, this works only with `ListView` controls, so it has limited use. However, when it does work for your application, it can save on some coding.



### Tip

You can create `ListView` headers and footers using `ListView.FixedViewInfo` with the `ListView` methods `addHeaderView()` and `addFooterView()`.

## Organizing Screens with Tabs

The Android SDK has a flexible way to provide a tab interface to the user. Much like `ListView` and `ListActivity`, there are two ways to create tabbing on the Android platform. You can either use the `TabActivity`, which simplifies a screen with tabs, or you can create your own tabbed screens from scratch. Both methods rely on the `TabHost` control.



### Warning

The Eclipse Layout Resource editor does not display `TabHost` controls properly in design mode. In order to design this kind of layout, you should stick to the XML layout mode. You must use the Android emulator or an Android device to view the tabs.

## Using TabActivity

A screen layout with tabs consists of a `TabActivity` and a `TabHost`. The `TabHost` consists of `TabSpecs`, a nested class of `TabHost`, which contains the tab information including the tab title and the contents of the tab. The contents of the tab can either be a predefined `View`, an `Activity` launched through an `Intent` object, or a factory-generated `View` using the `TabContentFactory` interface.

Tabs aren't as complex as they might sound at first. Each tab is effectively a container for a `View`. That `View` can come from any `View` that is ready to be shown, such as an XML layout file. Alternatively, that `View` can come from launching an `Activity`. The following example demonstrates each of these methods using `View` controls and `Activity` objects created in the previous examples of this chapter:

```
public class TabLayout
    extends TabActivity
    implements android.widget.TabHost.TabContentFactory {
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    TabHost tabHost = getTabHost();
    LayoutInflater.from(this).inflate(
        R.layout.example_layout,
        tabHost.getTabContentView(), true);
    tabHost.addTab(tabHost.newTabSpec("tab1")
        .setIndicator("Grid").setContent(
            new Intent(this, GridLayout.class)));
    tabHost.addTab(tabHost.newTabSpec("tab2")
        .setIndicator("List").setContent(
```

```
        new Intent(this, List.class)));
    tabHost.addTab(tabHost.newTabSpec("tab3")
        .setIndicator("Basic").setContent(
            R.id.two_texts));
    tabHost.addTab(tabHost.newTabSpec("tab4")
        .setIndicator("Factory").setContent(
            this));
}

public View createTabContent(String tag) {
    if (tag.compareTo("tab4") == 0) {
        TextView tv = new TextView(this);
        Date now = new Date();
        tv.setText("I'm from a factory. Created: "
            + now.toString());
        tv.setTextSize((float) 24);
        return (tv);
    } else {
        return null;
    }
}
}
```

This example creates a tabbed layout view with four tabs on it, as shown in Figure 9.9. The first tab is from the recent `GridView` example. The second tab is from the `ListView` example before that. The third tab is the basic layout with two `TextView` controls, fully defined in an XML layout file, as previously demonstrated. Finally, the fourth tab is created with a factory.

The first action is to get the `TabHost` instance. This is the object that enables us to add `Intent` objects and `View` identifiers for drawing the screen. A `TabActivity` provides a method to retrieve the current `TabHost` object.

The next action is only loosely related to tab views. The `LayoutInflater` is used to turn the XML definition of a `View` into the actual `View` controls. This would normally happen when calling `setContentView()`, but we're not doing that. The use of the `LayoutInflater` is required for referencing the `View` controls by identifier, as is done for the third tab.

The code finishes up by adding each of the four tabs to the `TabHost` in the order that they will be presented. This is accomplished by multiple calls to the `addTab()` method of `TabHost`. The first two calls are essentially the same. Each one creates a new `Intent` with the name of an `Activity` that launches within the tab. These are the same `Activity` classes used previously for full-screen display. Even if the `Activity` isn't designed for full-screen use, this should work seamlessly.

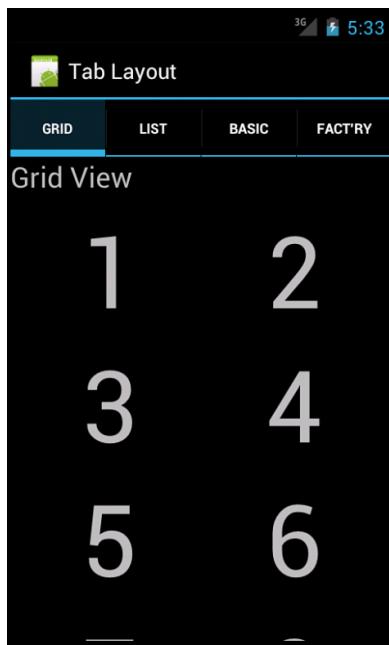


Figure 9.9 Four tabs displayed.

Next, on the third tab, a layout view is added using its identifier. In the preceding call to the `LayoutInflator`, the layout file also contains an identifier matching the one used here at the top level of a `LinearLayout` definition. This is the same one used previously to show a basic `LinearLayout` example. Again, there was no need to change anything in this view for it to display correctly in a tab.

Next, a tab referencing the content as the `TabActivity` class is added. This is possible because the class itself also implements `TabHost.TabContentFactory`, which requires implementing the `createTabContent()` method. The view is created the first time the user selects the tab, so no other information is needed here. The tag that creates this tab must be kept track of, though, as that's how the tabs are identified to the `TabHost`.

Finally, the method `createTabContent()` is implemented for use with the fourth tab. The first task here is to check the tag to see if it's the one kept track of for the fourth tab. When that is confirmed, an instance of the `TextView` control is created and a text string assigned to it, which contains the current time. The size of the text is set to 24 pixels. The time stamp used in this string can be used to demonstrate when the view is created and that it's not re-created by simply changing tabs.

The flexibility of tabs that Android provides is great for adding navigation to an application that has a bunch of views already defined. Few changes, if any, need to be made to existing `View` and `Activity` objects for them to work within the context of a `TabHost`.



### Note

It is possible to design tabbed layouts without using the `TabActivity` class. However, this requires a bit of knowledge about the underpinnings of the `TabHost` and `TabWidget` controls. To define a set of tabs within an XML layout file without using `TabActivity`, begin with a `TabHost` (for example, `TabHost1`). This `TabHost` must have the ID `@android:id/tabhost`. Inside the `TabHost`, include a vertically oriented `LinearLayout` that must contain a `TabWidget` (which must have the ID `@android:id/tabs`) and a `FrameLayout` (which must have the ID `@android:id/tabcontent`). The contents of each tab are then defined within the `FrameLayout`.

After you've defined the `TabHost` properly in XML, you must load and initialize it using the `TabHost setup()` method on your activity's `onCreate()` method. First, you need to create a `TabSpec` for each tab, setting the tab indicator using the `setIndicator()` method and the tab content using the `setContent()` method. Next, add each tab using the `addTab()` method of the `TabHost`. Finally, you should set the default tab of the `TabHost`, using a method such as `setCurrentTabByTag()`.



### Note

In Android 3.0 and up, including Ice Cream Sandwich, the `ActionBar` control allows setting tabs, as well. Instead of loading views, these tabs allow the loading of `Fragments`. See <http://goo.gl/ZRR5j> for more information.

## Adding Scrolling Support

One of the easiest ways to provide vertical scrolling for a screen is by using the `ScrollView` (vertical scrolling) and `HorizontalScrollView` (horizontal scrolling) controls. Either control can be used as a wrapper container, causing all child view controls to have one continuous scroll bar. The `ScrollView` and `HorizontalScrollView` controls can have only one child, though, so it's customary to have that child be a layout, such as a `LinearLayout`, which then contains all the "real" child controls to be scrolled through.



### Tip

The code examples of scrolling in this section are provided in the `SimpleScrolling` application. The source code for the `SimpleScrolling` application is available for download on the book's websites.

Figure 9.10 shows a screen with and without a `ScrollView` control.

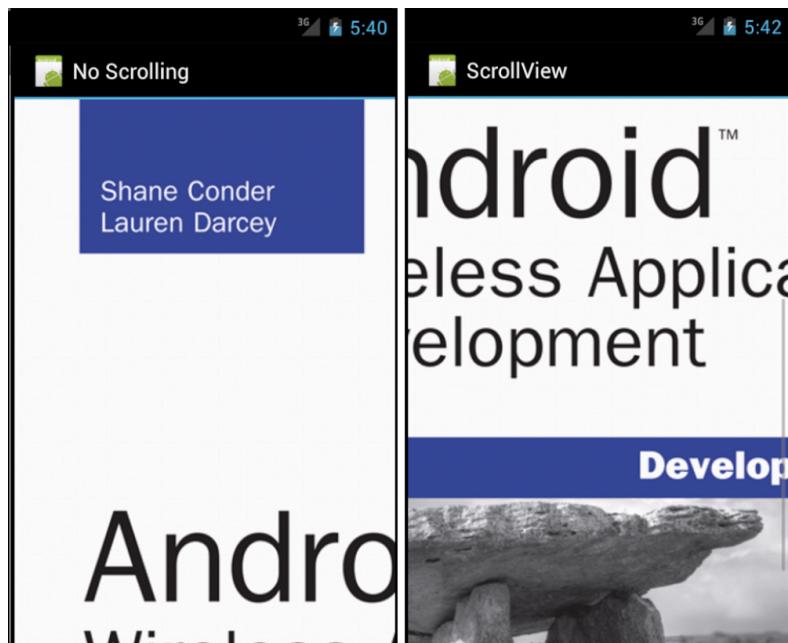


Figure 9.10 A screen without (left) and with (right) a ScrollView control.

## Exploring Other View Containers

Many other user interface controls are available within the Android SDK. Some of these controls are listed here:

- **Switchers:** A `ViewSwitcher` control contains only two child view controls and only one of those is shown at a time. It switches between the two, animating as it does so. Primarily, the `ImageSwitcher` and `TextSwitcher` objects are used. Each one provides a way to set a new child view, either a `Drawable` resource or a text string, and then animates from what is displayed to the new contents.
- **Sliding drawer:** Another view container is the `SlidingDrawer` control. This control includes two parts: a handle and a container view. The user drags the handle open and the internal contents are shown; then the user can drag the handle shut and the content disappears. The `SlidingDrawer` can be used horizontally or vertically and is always used from within a layout representing the larger screen. This makes the `SlidingDrawer` especially useful for application configurations such as game controls. A user can pull out the drawer, pause the game, change some features, and then close the `SlidingDrawer` to resume the game.

## Summary

The Android SDK provides a number of powerful methods for designing usable and great-looking screens. This chapter introduced you to many of these. You first learned about many of the Android layout controls that can manage the placement of your controls on the screen. In many cases, these enable you to have a single screen design that works on most screen sizes and aspect ratios.

You then learned about other objects that contain views and how to group or place them on the screen in a particular way. These included such display paradigms as the tab, typically used in a similar way that physical folder tabs are used, in addition to a variety of different controls for placing data on the screen in a readable and browsable way. You now have all the tools you need to develop applications with usable and exciting user interfaces.

## References and More Information

Android SDK Reference regarding the application `ViewGroup` class:

<http://d.android.com/reference/android/view/ViewGroup.html>

Android SDK Reference regarding the application `LinearLayout` class:

<http://d.android.com/reference/android/widget/LinearLayout.html>

Android SDK Reference regarding the application `RelativeLayout` class:

<http://d.android.com/reference/android/widget/RelativeLayout.html>

Android SDK Reference regarding the application `FrameLayout` class:

<http://d.android.com/reference/android/widget/FrameLayout.html>

Android SDK Reference regarding the application `TableLayout` class:

<http://d.android.com/reference/android/widget/TableLayout.html>

Android SDK Reference regarding the application `GridLayout` class:

<http://d.android.com/reference/android/widget/GridLayout.html>

Android SDK Reference regarding the application `ListView` class:

<http://d.android.com/reference/android/widget/ListView.html>

Android SDK Reference regarding the application `ListActivity` class:

<http://d.android.com/reference/android/app/ListActivity.html>

Android SDK Reference regarding the application `Gallery` class:

<http://d.android.com/reference/android/widget/Gallery.html>

Android SDK Reference regarding the application `GridView` class:

<http://d.android.com/reference/android/widget/GridView.html>

Android SDK Reference regarding the application `TabHost` class:

<http://d.android.com/reference/android/widget/TabHost.html>

Android SDK Reference regarding the application `TabActivity` class:

<http://d.android.com/reference/android/app/TabActivity.html>

Android Dev Guide: “Declaring Layout”:

<http://d.android.com/guide/topics/ui/declaring-layout.html>

*This page intentionally left blank*

# Working with Fragments

Traditionally, each screen within an Android application was tied to a specific Activity class. However, in Android 3.0 (Honeycomb), the concept of a user interface component called a Fragment was introduced. It was then included in the Android Support library for use with Android 1.6 (API Level 4) and up. Fragments decouple user interface behavior from a specific Activity lifecycle. Instead, Activity classes can mix and match user interface components to create more flexible user interfaces for the Android devices of the future.

## Understanding Fragments

Fragments were added to the Android SDK at a crucial time when Android consumers were experiencing an explosion in the variety of Android devices coming to market. We now see not just smartphones but other larger-screened devices such as tablets and televisions that run the platform. These larger devices come with substantially more screen real estate for developers to take advantage of. Your typical streamlined and elegant smartphone user interface often looks over-simplified on a tablet, for example. By incorporating Fragment components into your user interface design, you write one application that can be tailored to these different screen characteristics and orientations instead of different, tailored applications for different types of devices. This greatly improves code reuse, simplifies application testing needs, and makes publication and application package management much less cumbersome.

As we stated in the introduction to this chapter, the basic rule of thumb for developing Android applications used to be to have one Activity per screen of your application. This ties the underlying “task” functionality of an Activity class very directly to the user interface. However, as bigger device screens came along, this technique faced some issues. When you had more room on a single screen to do more, you had to implement separate Activity classes, with very similar functionality, to handle the cases where you wanted to provide more functionality on a given screen. Fragments help manage this problem by encapsulating screen functionality into reusable components that can be mixed and matched within Activity classes.

Let's look at a theoretical example. Let's say you have a traditional smartphone application with two screens. Perhaps it's an online news journal application. The first screen contains a `ListActivity` with a `ListView` control. Each item in the `ListView` represents an article available from the journal that you might want to read. When you click a specific article, you are sent to a new screen that displays the article contents in a `WebView` control. This traditional screen workflow is illustrated in Figure 10.1.

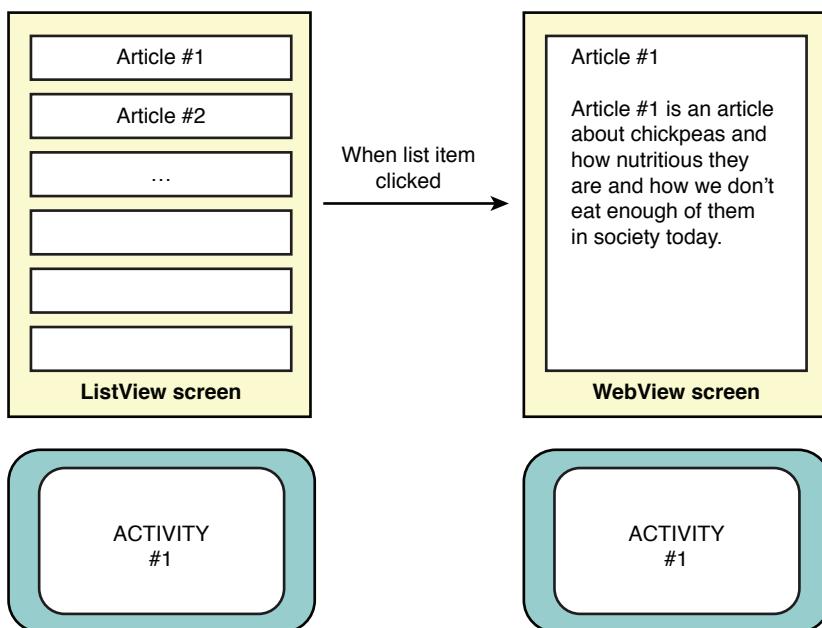


Figure 10.1 Traditional screen workflow without fragments.

This workflow works fine for small-screened smartphones, but it's a waste of all the space on a tablet or a television. Here, you might want to be able to peruse the article list and preview or read the article on the same screen. If we organize the `ListView` and the `WebView` screen functionality into two standalone `Fragment` components, then we can easily create a layout that includes both on the same screen when screen real estate allows, as shown in Figure 10.2.

## Understanding the Fragment Lifecycle

We discussed the `Activity` lifecycle back in Chapter 5, “Understanding the Anatomy of an Android Application.” Now let's look at how a `Fragment` fits into the mix. First of all, a `Fragment` must be hosted within an `Activity` class. It has its own lifecycle, but it is not a standalone component that can exist outside the context of an `Activity`.

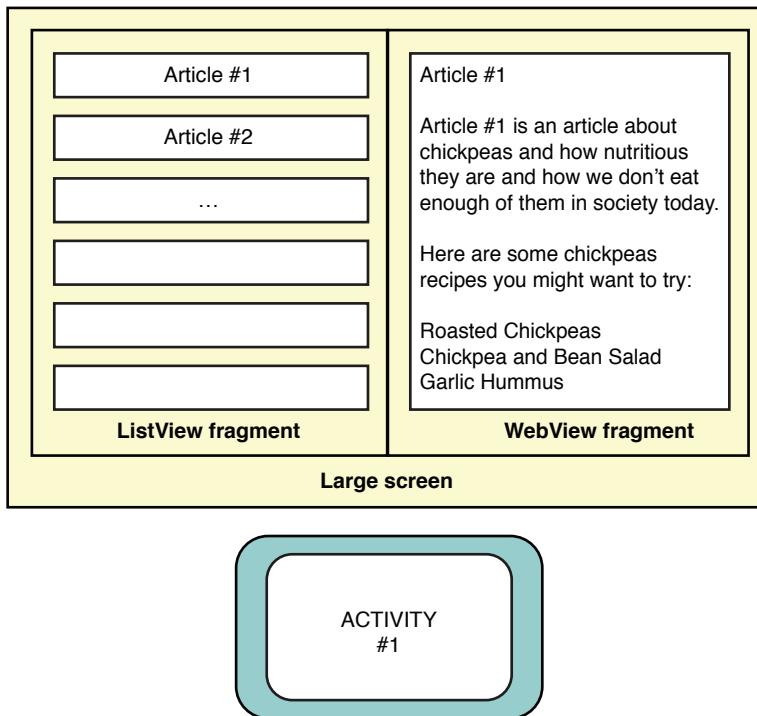


Figure 10.2 Improved screen workflow with fragments.

The responsibilities of the `Activity` class management are greatly simplified when the entire user interface state is moved off into individual fragments. `Activity` classes with only fragments in their layouts no longer need to spend a lot of time saving and restoring their state because the `Activity` object now keeps track of any `Fragment` that is currently attached automatically. The `Fragment` components themselves keep track of their own state using their own lifecycle. Naturally, you can mix fragments with `View` controls directly in an `Activity` class. The `Activity` class will be responsible for managing the `View` controls, as normal.

Instead, the `Activity` must focus on managing its `Fragment` classes. Coordination between an `Activity` and its `Fragment` components is facilitated by the `FragmentManager` (`android.app.FragmentManager`). The `FragmentManager` is acquired from the `getFragmentManager()` method, which is available within the `Activity` and `Fragment` classes.

## Defining Fragments

Fragment implementations that have been defined as regular classes within your application can be added to your layout resource files by using the `<fragment>` XML tag and

then loaded into your Activity using the standard `setContentView()` method, which is normally called in the `onCreate()` method of your Activity.

When you reference a Fragment class that you have defined in your application package in an XML layout file, use the `<fragment>` tag. This tag has a few important attributes. Specifically, you will need to set the `android:name` attribute of the fragment to the fully qualified Fragment class name. You will also need to give the item a unique identifier using the `android:id` attribute so that you can access that component programmatically, if needed. Like other XML layout controls, you still need to set the component's `layout_weight` and `layout_height` attributes like you would any other control in your layout. Here's a simple example of a `<fragment>` layout reference that refers to a class called `FieldNoteListFragment`, which is defined as a `.java` class in the package.

```
<fragment
    android:name="com.androidbook.simplefragments.FieldNoteListFragment"
    android:id="@+id/list"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

## Managing Fragment Modifications

As you can see, when you have multiple Fragment components on a single screen, within a single Activity, you often have user interaction on one Fragment (such as our news ListView Fragment) causing the Activity to update another Fragment (such as our article WebView Fragment). An update or modification to a Fragment is performed using a `FragmentTransaction` (`android.app.FragmentTransaction`). A number of different actions can be applied to a Fragment using a `FragmentTransaction` operation, such as the following:

- A Fragment can be attached or reattached to the parent Activity.
- A Fragment can be hidden and unhidden from view.

Perhaps at this point you are wondering how the Back button fits into this new Fragment-based user interface design. Well, now the parent Activity class has its own back stack. As the developer, you can decide which `FragmentTransaction` operations are worth storing in the back stack and which are not by using the `addToBackStack()` method of the `FragmentManager` object. For example, with our news application example, we might want each of the articles displayed in the WebView Fragment to be added to the parent Activity class's back stack so that if the user hits the Back button, he traverses the articles he has already read before backing out of the Activity entirely.

## Attaching and Detaching Fragments with Activities

After you have a Fragment that you want to include within your Activity class, the lifecycle of the Fragment comes into play. The following callback methods are important to managing the lifecycle of a Fragment, as it is created and then destroyed when it is no longer used. Many of these lifecycle events mirror those in the Activity lifecycle:

- The `onAttach()` callback method is called when a `Fragment` is first attached to a specific `Activity` class.
- The `onCreate()` callback method is called when a `Fragment` is first being created.
- The `onCreateView()` callback method is called when the user interface layout, or view *hierarchy*, associated with the `Fragment` should be created.
- The `onActivityCreated()` callback method will inform the `Fragment` when its parent `Activity` class's `onCreate()` method has completed.
- The `onStart()` callback method is called when the `Fragment`'s user interface becomes visible, but not yet active.
- The `onResume()` callback method makes the `Fragment`'s user interface active for interaction after the `Activity` has resumed or the `Fragment` was updated using a `FragmentTransaction`.
- The `onPause()` callback method is called when the parent `Activity` is paused, or the `Fragment` is being updated by a `FragmentTransaction`. It indicates that the `Fragment` is no longer active or in the foreground.
- The `onStop()` callback method is called when the parent `Activity` is stopped, or the `Fragment` is being updated by a `FragmentTransaction`. It indicates the `Fragment` is no longer visible.
- The `onDestroyView()` callback method is called to clean up any user interface layout, or view hierarchy resources, associated with the `Fragment`.
- The `onDestroy()` callback method is called to clean up any other resources associated with the `Fragment`.
- The `onDetach()` callback method is called just before the `Fragment` is detached from the `Activity` class.

## Working with Special Types of Fragments

If you recall from Chapter 9, “Designing User Interfaces with Layouts,” there are a number of special `Activity` classes for managing certain common types of user interfaces. For example, the `ListActivity` class simplifies the creation of an `Activity` that manages a `ListView` control. Similarly, the `PreferenceActivity` class simplifies the creation of an `Activity` to manage shared preferences. And as we saw in our theoretical news reader application example, we often want to use user interface controls such as `ListView` and `WebView` within our fragment components.

Because fragments are meant to decouple this user interface functionality from the `Activity` class, you'll now find equivalent `Fragment` subclasses that perform this functionality instead. Some of the specialty `Fragment` classes you'll want to familiarize yourself with include the following:

- **ListFragment (`android.app.ListFragment`)**: Much like a `ListAdapter`, this Fragment class hosts a `ListView` control.
- **PreferenceFragment (`android.preference.PreferenceFragment`)**: Much like a `PreferenceActivity`, this Fragment class lets you easily manage user preferences.
- **WebViewFragment (`android.webkit.WebViewFragment`)**: This type of Fragment hosts a `WebView` control to easily render web content. Your application will still need the `android.permission.INTERNET` permission to access the Internet.
- **DialogFragment (`android.app.DialogFragment`)**: Decoupling user interface functionality from your `Activity` classes means you won't want your dialogs managed by the `Activity` either. Instead, you can use this class to host and manage Dialog controls as Fragments. Dialogs can be traditional pop-ups or embedded. We discuss dialogs in detail in Chapter 11, "Working with Dialogs."



#### Note

You may have noticed that `TabActivity`, the helper class for working with the `TabHost` control, is not listed as a Fragment class. If you are simply using `TabHost` without the `TabActivity` helper class, you can easily move this into a Fragment. However, if you are using `TabActivity`, then when you move to a Fragment-based application design, you'll want to look over how the action bars work, which allow you to add tabs. For more information, see the Android SDK documentation for the `TabActivity` (`android.app.TabActivity`), `ActionBar` (`android.app.ActionBar`), and `ActionBar.Tab` (`android.app.ActionBar.Tab`) classes. We discuss action bars in detail in *Android Wireless Application Development Volume II: Advanced Topics*.

## Designing Fragment-Based Applications

At the end of the day, Fragment-based applications are best learned by example. Therefore, let's work through a fairly straightforward example to help nail down the many concepts we have discussed thus far in the chapter. To keep things simple, we will target a specific version of the Android platform: Android 3.2. However, you will soon find that you can also create Fragment-based applications for almost any device by using the Android Support Package.



#### Tip

Many of the code examples provided in this section are taken from the `SimpleFragments` application. The source code for the `SimpleFragments` application is provided for download on the book's website.

We (the authors) are big travelers. When we went to Africa, we took tons of pictures and wrote up a bunch of information about the different animals we saw in the wild on our blog. We called it our “African Field Notes” (<http://goo.gl/hA0fh>). Let’s make a simple application with a `ListView` of wild animal names. Clicking a `ListView` item will load a `WebView` control and display the specific blog post associated with that animal. To keep things simple, we’ll store our list of animals and blog URLs in string array resources. (See the sample code for a complete implementation.)

So how will our fragments work? We will use a `ListFragment` for the list of animals and a `WebViewFragment` to display each blog post. In portrait mode, we will display one fragment per screen, requiring two `Activity` classes, as shown in Figure 10.3.

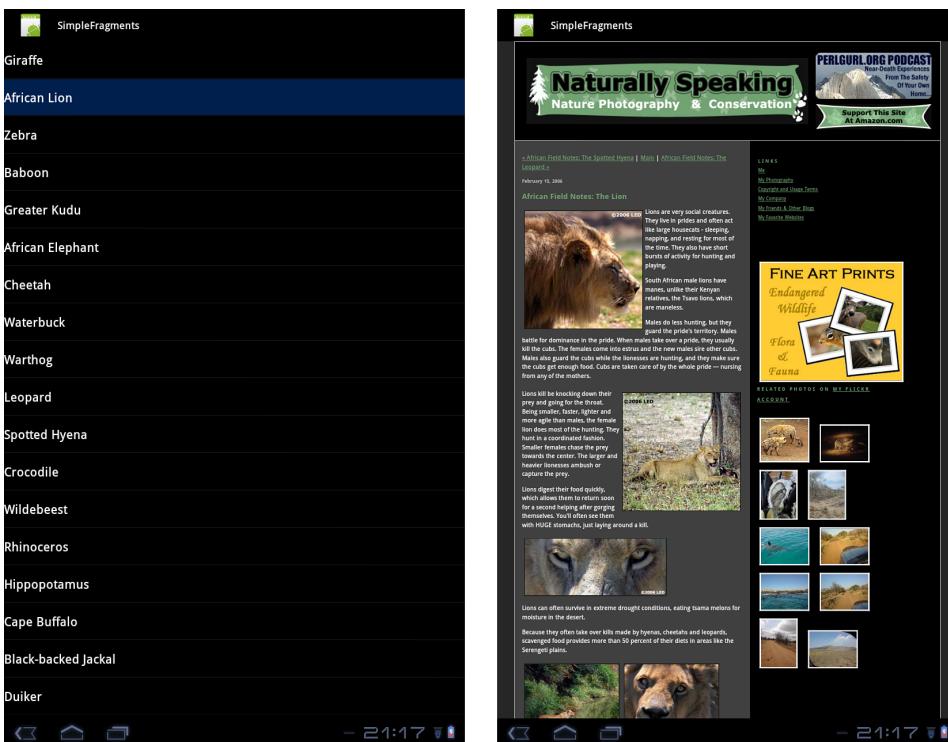


Figure 10.3 One fragment per activity/screen.

In landscape mode, we will display both fragments on the same screen within the same `Activity` class, as shown in Figure 10.4.

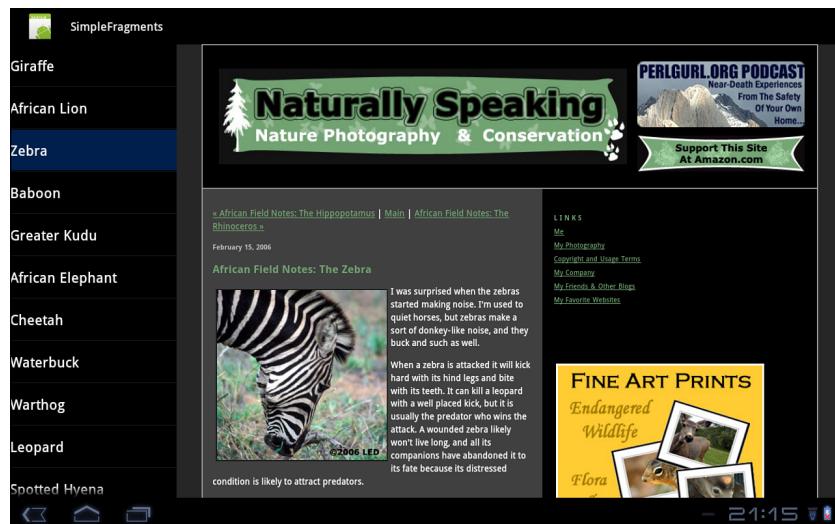


Figure 10.4 Both fragments in a single activity/screen.

### Implementing a ListFragment

Let's begin by defining a custom `ListFragment` class called `FieldNoteListFragment` to host our wild animal names. This class will need to determine whether the second Fragment, the `FieldNoteWebViewFragment`, should be loaded or if `ListView` clicks should simply cause the `FieldNoteViewActivity` to be launched:

```
public class FieldNoteListFragment extends ListFragment implements
    FragmentManager.OnBackStackChangedListener {

    private static final String DEBUG_TAG = "FieldNoteListFragment";
    int mCurPosition = -1;
    boolean mShowTwoFragments;

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

        getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);

        String[] fieldNotes = getResources().getStringArray(
            R.array.fieldnotes_array);
        setListAdapter(new ArrayAdapter<String>(getActivity(),
            android.R.layout.simple_list_item_activated_1, fieldNotes));
    }
}
```

```
View detailsFrame = getActivity().findViewById(R.id.fieldentry);
mShowTwoFragments = detailsFrame != null
    && detailsFrame.getVisibility() == View.VISIBLE;

if (savedInstanceState != null) {
    mCurPosition = savedInstanceState.getInt("curChoice", 0);
}

if (mShowTwoFragments == true || mCurPosition != -1) {
    viewAnimalInfo(mCurPosition);
}

getFragmentManager().addOnBackStackChangedListener(this);
}

@Override
public void onBackStackChanged() {
    FieldNoteWebViewFragment details =
        (FieldNoteWebViewFragment) getFragmentManager()
            .findFragmentById(R.id.fieldentry);
    if (details != null) {
        mCurPosition = details.getShownIndex();
        getListView().setItemChecked(mCurPosition, true);

        if (!mShowTwoFragments) {
            viewAnimalInfo(mCurPosition);
        }
    }
}

@Override
public void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    outState.putInt("curChoice", mCurPosition);
}

@Override
public void onListItemClick(ListView l, View v, int position, long id) {
    viewAnimalInfo(position);
}

void viewAnimalInfo(int index) {
    mCurPosition = index;
    if (mShowTwoFragments == true) {
        // Check what fragment is currently shown, replace if needed.
        FieldNoteWebViewFragment details =
```

```
(FieldNoteWebViewFragment) getFragmentManager()
    .findFragmentById(R.id.fieldentry);
if (details == null || details.getShownIndex() != index) {

    FieldNoteWebViewFragment newDetails = FieldNoteWebViewFragment
        .newInstance(index);

    FragmentManager fm = getFragmentManager();
    FragmentTransaction ft = fm.beginTransaction();
    ft.replace(R.id.fieldentry, newDetails);
    if (index != -1) {
        String[] fieldNotes = getResources().getStringArray(
            R.array.fieldnotes_array);
        String strBackStackTagName = fieldNotes[index];
        ft.addToBackStack(strBackStackTagName);
    }
    ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
    ft.commit();
}

} else {
    Intent intent = new Intent();
    intent.setClass(getActivity(), FieldNoteViewActivity.class);
    intent.putExtra("index", index);
    startActivity(intent);
}
}
```

Most of the `Fragment` control's initialization happens in the `onActivityCreated()` callback method so that we only initialize the `ListView` once. We then check to see which display mode we want to be in by checking to see if our second component is defined in the layout. Finally, we leave the display details to the helper method called `viewAnimalInfo()`, which is also called whenever an item in the `ListView` control is clicked.

The logic for the `viewAnimalInfo()` method takes into account both display modes. If the device is in portrait mode, the `FieldNoteViewActivity` is launched via an `Intent`. However, if the device is in landscape mode, we have some `Fragment` finagling to do.

Specifically, the `FragmentManager` is used to find the existing `FieldNoteWebView` Fragment by its unique identifier (`R.id.fieldentry`, as defined in the layout resource file). Then, a new `FieldNoteWebViewFragment` instance is created for the new animal blog post being requested. Next, a `FragmentTransaction` is started, in which the existing `FieldNoteWebViewFragment` is replaced with the new one. We put the old one on

the back stack so that the Back button works nicely, set the transition animation to fade between the blog entries, and commit the transaction, thus causing the screen to update asynchronously.

Finally, we can monitor the back stack with a call to the `addOnBackStackChangedListener()` method. The callback, `onBackStackChanged()`, updates the list to the current selected item. This provides a robust way to keep the `ListView` item selection synchronized with the currently displayed `Fragment` both when adding a new `Fragment` to the back stack and when removing one, such as when the user presses the Back button.

### Implementing a `WebViewFragment`

Next, we create a custom `WebViewFragment` class called `FieldNoteWebViewFragment` to host the blog entries related to each wild animal. This `Fragment` class does little more than determine which blog entry URL to load and then load it in the `WebView` control.

```
public class FieldNoteWebViewFragment extends WebViewFragment {

    private static final String DEBUG_TAG = "FieldNoteWebViewFragment";

    public static FieldNoteWebViewFragment newInstance(int index) {
        Log.v(DEBUG_TAG, "Creating new instance: " + index);
        FieldNoteWebViewFragment fragment =
            new FieldNoteWebViewFragment();

        Bundle args = new Bundle();
        args.putInt("index", index);
        fragment.setArguments(args);
        return fragment;
    }

    public int getShownIndex() {
        int index = -1;
        Bundle args = getArguments();
        if (args != null) {
            index = args.getInt("index", -1);
        }
        if (index == -1) {
            Log.e(DEBUG_TAG, "Not an array index.");
        }
        return index;
    }
}
```

```

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

    String[] fieldNoteUrls = getResources().getStringArray(
        R.array.fieldnoteurls_array);
    int fieldNoteUrlIndex = getShownIndex();

    WebView webview = getWebView();
    webview.setPadding(0, 0, 0, 0);
    webview.getSettings().setLoadWithOverviewMode(true);
    webview.getSettings().setUseWideViewPort(true);

    if (fieldNoteUrlIndex != -1) {
        String fieldNoteUrl = fieldNoteUrls[fieldNoteUrlIndex];
        webview.loadUrl(fieldNoteUrl);
    }
    else
    {
        webview.loadUrl("http://www.perlgurl.org/archives/
➥photography/special_assignments/african_field_notes/");
    }
}
}

```

Most of the Fragment control's initialization happens in the `onActivityCreated()` callback method so that we only initialize the `WebView` once. The default configuration of the `WebView` control doesn't look so pretty, so we make some configuration changes, remove the padding around the control, and set some settings to make the browser fit nicely in the screen area provided. If we've received a request for a specific animal to load, we look up the URL and load it; otherwise, we load the "default" front page of the field notes blog.

## Defining the Layout Files

Now that you've implemented your Fragment classes, you can place them in the appropriate layout resource files. You'll need to create two layout files. In landscape mode, you'll want a single `main.xml` layout file to host both Fragment components. In portrait mode, you'll want a comparable layout file that only hosts the `ListFragment` you implemented. The `WebViewFragment` you implemented will have a user interface generated at runtime.

Let's start with the landscape mode layout resource, called `/res/layout-land/main.xml`. Note that we store this `main.xml` resource file in a special resource directory for landscape mode use only. We discuss how to store alternative resources in this way in depth in Chapter 15, "Designing Compatible Applications." For now, suffice it to say that this layout will be automatically loaded whenever the device is in landscape mode.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment
        android:name="com.androidbook.simplefragments.FieldNote
    <ListFragment"
        android:id="@+id/list"
        android:layout_weight="1"
        android:layout_width="0dp"
        android:layout_height="match_parent" />
    <FrameLayout
        android:id="@+id/fieldentry"
        android:layout_weight="4"
        android:layout_width="0dp"
        android:layout_height="match_parent" />
</LinearLayout>
```

Here we have a fairly straightforward `LinearLayout` control with two child controls. One is a static `Fragment` component that references the custom `ListFragment` class you implemented. For the second region where we want to put the `WebViewFragment`, we include a `FrameLayout` region that we will replace with our specific `FieldNoteWebViewFragment` instance programmatically at runtime.

### Tip

When dealing with `Fragment` components that will be updated via add or replace (dynamic), do not mix them with `Fragment` components instantiated via the layout (static). Instead, use a placeholder element such as a `FrameLayout`, as in the sample code. Dynamic `Fragment` components and the static ones defined using `<fragment>` from the layout do not mix well with the fragment transaction manager or with the back stack.

The resources stored in the normal layout directory will be used whenever the device is not in landscape mode (in other words, portrait mode). Here we need to define two layout files. First, let's define our static `ListFragment` in its own `/res/layout/main.xml` file. It looks much like the previous version, without the second `FrameLayout` control:

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

```

<fragment
    android:name="com.androidbook.simplefragments.
    ↪FieldNoteListFragment"
    android:id="@+id/list"
    android:layout_weight="1"
    android:layout_width="0dp"
    android:layout_height="match_parent" />
</LinearLayout>

```

## Defining the Activity Classes

You're almost done. Now you need to define your `Activity` classes to host your `Fragment` components. You'll need two `Activity` classes: a primary class and a secondary one that is only used to display the `FieldNoteWebViewFragment` when in portrait mode. Let's call the primary `Activity` class `SimpleFragmentsActivity` and the secondary `Activity` class `FieldNoteViewActivity`.

As mentioned earlier, moving all your user interface logic to `Fragment` components greatly simplifies your `Activity` class implementation. For example, here is the complete implementation for the `SimpleFragmentsActivity` class:

```

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

```

Yup. That's it. The `FieldNoteViewActivity` class is only slightly more interesting:

```

public class FieldNoteViewActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        if (getResources().getConfiguration().orientation ==
            Configuration.ORIENTATION_LANDSCAPE) {
            finish();
            return;
        }

        if (savedInstanceState == null) {
            FieldNoteWebViewFragment details = new FieldNoteWebViewFragment();
            details.setArguments(getIntent().getExtras());

            FragmentManager fm = getFragmentManager();
            FragmentTransaction ft = fm.beginTransaction();

```

```
        ft.add(android.R.id.content, details);
        ft.commit();
    }
}
```

Here we check that we're in the appropriate orientation to be using this `Activity`. Then we create an instance of the `FieldNoteWebViewFragment` and programmatically add it to the `Activity`, generating its user interface at runtime by adding it to the `android.R.id.content` view, which is the root view of any `Activity` class. That's all that's needed to implement this simple sample application with `Fragment` components.

## Using the Android Support Package

Fragments are so important to the future of the Android platform that the Android team provided a compatibility library so that developers can update their legacy applications as far back as Android 1.6, if they so chose. This library was originally called the Compatibility Package, and is now called the Android Support Package.

### Adding Fragment Support to Legacy Applications

The choice of whether or not to update older applications is a personal one for the development team. Non-fragment applications should continue to function for the foreseeable future without error, mostly due to the Android team's continued policy of supporting legacy applications as best as possible when new platform versions are released. Here are some considerations for developers with legacy applications who are considering whether or not to revise their existing code:

- Leave your legacy application as is, and the ramifications are not catastrophic. Your application will not be using the latest and greatest features that the Android platform has to offer (and users will notice this), but it should continue to run as well as it always has without any additional work on your part. If you have no plans to update or upgrade your old applications, this may very well be a reasonable choice. The potentially inefficient use of screen space may be problematic, but should not create new errors.
- If your application has a lot of market traction and you've continued to update it as the Android platform has matured, you're more likely to want to consider the Android Support Package. Your users may demand it. You can certainly continue to support your legacy application and create a separate new-and-improved version that uses the new platform features, but this means organizing and managing different source code branches, different application packages, and complicates application publication and reporting, not to mention maintenance. Better to revise your existing application to use the Android Support Package and do your best to keep your single codebase manageable. The size and resources of your organization may be a contributing factor to the decision here.

- Just because you start using the Android Support Package in your applications does not mean you have to implement every new feature (fragments, loaders, and so on) immediately. You can simply pick and choose the features that make the most sense for your application and add others over time via application updates when your team has the resources and inclination.
- Choosing to not update your code to new controls could leave your legacy application looking dated compared to other applications. If your application is already completely customized and isn't using stock controls—often the case with games and other highly graphical apps—then your application may not need updating. If, however, you conform to stock system controls, look, and feel, it may be more important for your application to get a fresh look.

## Using Fragments in New Applications Targeting Older Platforms

If you're just starting to develop a new application and plan to target some of the older platform versions, then incorporating fragments into your design is a much easier decision. If you're just starting a project, there's little reason not to use them and quite a few reasons why you should.

- Regardless of what devices and platforms you are targeting now, there will be new ones in the future that you cannot foresee. Fragments give you the flexibility to easily adjust your user interface screen workflows without rewriting or retesting all your application code.
- Incorporating the Android Support Package into your applications early means that if other important platform features are added later, you'll easily be able to update the libraries and start using them.
- By using the Android Support Package, your application will not show its age nearly as quickly since you will be incorporating the newer features of the platform and providing them to users on older platforms.

## Linking the Android Support Package to Your Project

The Android Support Package is simply a set of static support libraries (available as a .jar file) that you can link to your Android application and use. You can download the Android Support Package using the Android SDK Manager and then add it to the projects of your choice. It is an optional package and not linked by default. Android Support Packages are versioned like everything else, and they are updated occasionally with new features—and more importantly, bug fixes.



### Tip

You can find out more about the latest version package at the Android Developer website: <http://d.android.com/sdk/compatibility-library.html>.

There are actually two Android Support Packages at this time: v4 and v13. The v4 package aims to provide new classes introduced in Honeycomb and beyond to platform versions as far back as API Level 4 (Android 1.6). This is the package you want to use when supporting your legacy applications. The v13 package provides more efficient implementations of some items, such as the `FragmentPagerAdapter`, when running on API Level 13 and later. If you're targeting API Level 13 or later, use this package instead. Be aware that parts of the package that are part of the platform are not available in this package; they aren't needed.

To use the Android Support Package with your application, take the following steps:

1. Use the Android SDK Manager to download the Android Support Package (formerly the Compatibility Package).
2. Find your project in the Package Explorer or Project Explorer.
3. Right-click the project and choose Android Tools, Add Compatibility Library.... The most updated library will be downloaded, and your project settings will be modified to use the newest library.
4. Begin using the APIs available as part of the Android Support Package. For example, to create a class extending `FragmentActivity`, you'd need to import `android.support.v4.app.FragmentActivity`.



### Note

A few differences exist between the APIs used by the Android Support Package and those found in the later versions of the Android SDK. However, there are some renamed classes to avoid name collisions, and not all classes and features are currently incorporated into the Android Support Package.

## Summary

Fragments were introduced into the Android SDK to help address the different types of device screens that application developers need to target now and in the future. A `Fragment` is simply a self-contained chunk of a user interface, with its own lifecycle, that can be independent of a specific `Activity` class. Fragments must be hosted within `Activity` classes, but they give the developer a lot more flexibility when it comes to breaking screen workflow into components that can be mixed and matched in different ways, depending on the screen real estate available on the device. Fragments were introduced in Android 3.0, but legacy applications can use them if they take advantage of the Android Support Package, which allows applications that target API Level 4 (Android 1.6) and higher to use these more recent additions to the Android SDK.

## References and More Information

Android SDK Reference regarding the application `Fragment` class:

<http://d.android.com/reference/android/app/Fragment.html>

Android SDK Reference regarding the application `ListFragment` class:

<http://d.android.com/reference/android/app/ListFragment.html>

Android SDK Reference regarding the application `PreferenceFragment` class:

<http://d.android.com/reference/android/preference/PreferenceFragment.html>

Android SDK Reference regarding the application `WebViewFragment` class:

<http://d.android.com/reference/android/webkit/WebViewFragment.html>

Android SDK Reference regarding the application `DialogFragment` class:

<http://d.android.com/reference/android/app/DialogFragment.html>

Android Dev Guide: “Fragments”:

<http://d.android.com/guide/topics/fundamentals/fragments.html>

Android Developers Blog: “The Android 3.0 Fragments API”:

<http://android-developers.blogspot.com/2011/02/android-30-fragments-api.html>

Developer.com: “Create Flexible Android UIs with Fragments”:

<http://www.developer.com/ws/create-flexible-android-uis-with-fragments.html>

# Working with Dialogs

Android application user interfaces need to be elegant and easy to use. One important technique developers can use is to implement dialogs to inform the user or allow the user to perform actions such as edits without redrawing the main screen. In this chapter, we discuss how to incorporate dialogs into your applications.

## Choosing Your Dialog Implementation

The Android platform is growing and changing quickly. New revisions of the Android SDK are released on a frequent basis. This means that developers are always struggling to keep up with the latest that Android has to offer. Right now the Android platform is in a period of transition from a traditional smartphone platform to a “smart-device” platform that will support a much wider variety of devices, such as tablets, TVs, and toasters. To this end, one of the most important additions to the platform is the concept of the fragment. We discussed fragments in detail in the previous chapter, but they have wide ramifications in terms of Android application user interface design. One area of application design that has received an overhaul during this transition is the way in which dialogs are implemented.

So what does this mean for developers? It means that there are now two methods for incorporating dialogs into your application—the legacy method and the method recommended for developers moving forward:

- Using the legacy method, which has existed since the first Android SDK was released, an `Activity` class manages its dialogs in a dialog pool. Dialogs are created, initialized, updated, and destroyed using `Activity` class callback methods. Dialogs are not shared among activities. This type of dialog implementation works for all versions of the Android platform; however, many of the methods used in this type of solution have been deprecated as of API Level 13 (Android 3.2). If your application `Activity` classes are not using, and do not plan to use, the `Fragment` APIs, then this method may be the most straightforward to implement, despite being deprecated and perhaps not receiving future support, bug fixes, or exhaustive testing.

- Using the new fragment-based method, which was introduced in API Level 11 (Android 3.0), dialogs are managed using the `FragmentManager` class (`android.app.FragmentManager`). Dialogs become a special type of `Fragment` that must still be used within the scope of an `Activity` class, but its lifecycle is managed like any other `Fragment`. This type of dialog implementation works with the newest versions of the Android platform, but is not backward compatible with older devices unless you incorporate the latest Android Support Package into your application to gain access to these new classes for use with older Android SDKs. However, this is likely the best choice for moving forward with the Android platform.



### Note

Unlike with some other platforms, which routinely remove deprecated methods after a few releases, deprecated methods within the Android SDK can normally be used safely for the foreseeable future, as necessary. That said, developers should understand the ramifications of using deprecated methods and techniques, which may include difficulty in upgrading application functionality to use the latest SDK features later on, slower performance as newer features are streamlined and legacy ones are left as is, and the possibility of the application “showing its age.” Deprecated methods are also unlikely to receive any sort of fixes or updates.

We will cover both methods [in this](#) chapter, given that most applications on the Android Market today are still using the deprecated method. If you’re maintaining legacy applications, you’ll need to understand this method. However, if you are developing new applications or updating existing applications to use the latest technologies the Android SDK has to offer, we highly recommend implementing the newer `Fragment`-based method and using the Android Support Package to support older versions of the Android platform.

## Exploring the Different Types of Dialogs

Regardless of which way you implement them, a number of different dialog types are available within the Android SDK. Each type has a special function that most users should be somewhat familiar with. The dialog types available as part of the Android SDK include the following:

- **Dialog**: The basic class for all `Dialog` types. A basic `Dialog` (`android.app.Dialog`) is shown in the top left of Figure 11.1.
- **AlertDialog**: A `Dialog` with one, two, or three `Button` controls. An `AlertDialog` (`android.app.AlertDialog`) is shown in the top center of Figure 11.1.
- **CharacterPickerDialog**: A `Dialog` for choosing an accented character associated with a base character. A `CharacterPickerDialog`

(`android.text.method.CharacterPickerDialog`) is shown in the top right of Figure 11.1.

- **DatePickerDialog:** A Dialog with a DatePicker control. A `DatePickerDialog` (`android.app.DatePickerDialog`) is shown in the bottom left of Figure 11.1.
- **ProgressDialog:** A Dialog with a determinate or indeterminate ProgressBar control. An indeterminate `ProgressDialog` (`android.app ProgressDialog`) is shown in the bottom center of Figure 11.1.
- **TimePickerDialog:** A Dialog with a TimePicker control. A `TimePickerDialog` (`android.app.TimePickerDialog`) is shown in the bottom right of Figure 11.1.

If none of the existing Dialog types is adequate, you can also create custom Dialog windows, with your specific layout requirements.



Figure 11.1 The different dialog types available in Android.

## Working with Dialogs: The Legacy Method

An Activity can use dialogs to organize information and react to user-driven events. For example, an activity might display a dialog informing the user of a problem or asking the user to confirm an action such as deleting a data record. Using dialogs for simple tasks helps keep the number of application activities manageable.

### Tip

Many of the code examples provided in this section are taken from the SimpleDialogs application. The source code for the SimpleDialogs application is provided for download on the book's website.

## Tracing the Lifecycle of a Dialog

Each Dialog must be defined within the Activity in which it is used. A Dialog may be launched once, or used repeatedly. Understanding how an Activity manages the Dialog lifecycle is important to implementing a Dialog correctly. Let's look at the key methods that an Activity must use to manage a Dialog:

- The `showDialog()` method is used to display a Dialog.
- The `dismissDialog()` method is used to stop showing a Dialog. The Dialog is kept around in the Activity's Dialog pool. If the Dialog is shown again using `showDialog()`, the cached version is displayed once more.
- The `removeDialog()` method is used to remove a Dialog from the Activity object's Dialog pool. The Dialog is no longer kept around for future use. If you call `showDialog()` again, the Dialog must be re-created.

Adding the Dialog to an Activity involves several steps:

1. Define a unique identifier for the Dialog within the Activity.
2. Implement the `onCreateDialog()` method of the Activity to return a Dialog of the appropriate type, when supplied the unique identifier.
3. Implement the `onPrepareDialog()` method of the Activity to initialize the Dialog as appropriate.
4. Launch the Dialog using the `showDialog()` method with the unique identifier.

### Defining a Dialog

A Dialog used and managed by an Activity must be defined in advance. Each Dialog has a special identifier (an integer). When the `showDialog()` method is called, you pass in this identifier. At this point, the `onCreateDialog()` method is called and must return a Dialog of the appropriate type.

It is up to the developer to override the `onCreateDialog()` method of the Activity and return the appropriate Dialog for a given identifier. If an Activity has multiple Dialog windows, the `onCreateDialog()` method generally contains a switch statement to return the appropriate Dialog based on the incoming parameter—the Dialog identifier.

### Initializing a Dialog

Because a Dialog is often kept around by the Activity in its Dialog pool, it might be important to reinitialize a Dialog each time it is shown, instead of just when it is created the first time. For this purpose, you can override the `onPrepareDialog()` method of the Activity.

Whereas the `onCreateDialog()` method may only be called once for initial Dialog creation, the `onPrepareDialog()` method is called each time the `showDialog()` method is called, giving the Activity a chance to modify the Dialog before it is shown to the user.

## Launching a Dialog

You can display any dialog defined within an Activity by calling the `showDialog()` method of the `Activity` class and passing it a valid `Dialog` object identifier—one that will be recognized by the `onCreateDialog()` method.

## Dismissing a Dialog

Most types of dialogs have automatic dismissal circumstances. However, if you want to force a `Dialog` to be dismissed, simply call the `dismissDialog()` method and pass in the `Dialog` identifier.

## Removing a Dialog from Use

Dismissing a `Dialog` does not destroy it. If the `Dialog` is shown again, its cached contents are redisplayed. If you want to force an `Activity` to remove a `Dialog` from its pool and not use it again, you can call the `removeDialog()` method, passing in the valid `Dialog` identifier. Although not generally needed, a single-use, but resource heavy, dialog may provide some benefit when being removed.

Here's an example of a simple class called `SimpleDialogsActivity` that illustrates how to implement a simple `Dialog` control that is launched when a `Button` called `Button_AlertDialog` (defined in a layout resource) is clicked:

```
public class SimpleDialogsActivity extends Activity {

    static final int ALERT_DIALOG_ID = 1;

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

        // Handle Alert Dialog Button
        Button launchAlertDialog = (Button) findViewById(
            R.id.Button_AlertDialog);
        launchAlertDialog.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                showDialog(ALERT_DIALOG_ID);
            }
        });
    }

    @Override
    protected Dialog onCreateDialog(int id) {
        switch (id) {

            case ALERT_DIALOG_ID:
```

```

        AlertDialog.Builder alertDialog = new
            AlertDialog.Builder(this);
        alertDialog.setTitle("Alert Dialog");
        alertDialog.setMessage("You have been alerted.");
        alertDialog.setIcon(android.R.drawable.btn_star);
        alertDialog.setPositiveButton(android.R.string.ok,
            new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog,
                    int which) {
                    Toast.makeText(getApplicationContext(),
                        "Clicked OK!", Toast.LENGTH_SHORT).show();
                    return;
                }
            });
        return alertDialog.create();
    }
    return null;
}

@Override
protected void onPrepareDialog(int id, Dialog dialog) {
    super.onPrepareDialog(id, dialog);
    switch (id) {
        case ALERT_DIALOG_ID:
            // No extra configuration needed
            return;
    }
}
}

```

The full implementation of this `AlertDialog`, as well as many other types of dialogs, can be found in the sample code provided on the book's website.

## Working with Custom Dialogs

When the dialog types do not suit your purpose exactly, you can create a custom Dialog. One easy way to create a custom Dialog is to begin with an `AlertDialog` and use an `AlertDialog.Builder` class to override its default layout. In order to create a custom Dialog this way, the following steps must be performed:

1. Design a custom layout resource to display in the `AlertDialog`.
2. Define the custom Dialog identifier in the `Activity`.
3. Update the `Activity`'s `onCreateDialog()` method to build and return the appropriate custom `AlertDialog`. You should use a `LayoutInflater` to inflate the custom layout resource for the Dialog.
4. Launch the Dialog using the `showDialog()` method.

## Working with Dialogs: The Fragment Method

Moving forward, most `Activity` classes should be “fragment aware.” This means that you’ll want to decouple your `Dialog` management from the `Activity` and move it into the realm of fragments. There is a special subclass of `Fragment` called a `DialogFragment` (`android.app.DialogFragment`) that can be used for this purpose.

### Tip

Many of the code examples provided in this section are taken from the `SimpleFragDialog` application. The source code for the `SimpleFragDialog` application is provided for download on the book’s website.

Let’s look at a quick example of how you might implement a simple `AlertDialog` that behaves much like the legacy `AlertDialog` discussed earlier in this chapter. In order to show an advantage of using the new fragment-based dialog technique, we pass some data to the dialog that demonstrates multiple instances of the `DialogFragment` class running within a single `Activity`. To begin, you need to implement your own `DialogFragment` class. This class simply needs to be able to return an instance of the object that is fully configured and needs to implement the `onCreateDialog` method, which returns the fully configured `AlertDialog`, much as it did using the legacy method. The following code is a full implementation of a simple `DialogFragment` that manages an `AlertDialog`:

```
public class MyAlertDialogFragment extends DialogFragment {  
  
    public static MyAlertDialogFragment  
        newInstance(String fragmentNumber) {  
        MyAlertDialogFragment newInstance = new MyAlertDialogFragment();  
        Bundle args = new Bundle();  
        args.putString("fragnum", fragmentNumber);  
        newInstance.setArguments(args);  
        return newInstance;  
    }  
  
    @Override  
    public Dialog onCreateDialog(Bundle savedInstanceState) {  
  
        final String fragNum = getArguments().getString("fragnum");  
  
        AlertDialog.Builder alertDialog = new AlertDialog.Builder(  
            getActivity());  
        alertDialog.setTitle("Alert Dialog");  
        alertDialog.setMessage("This alert brought to you by "  
            + fragNum );  
        alertDialog.setIcon(android.R.drawable.btn_star);  
    }  
}
```

```

        alertDialog.setPositiveButton(android.R.string.ok,
            new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int which) {
                    ((SimpleFragDialogActivity) getActivity())
                        .doPositiveClick(fragNum);
                    return;
                }
            });
        return alertDialog.create();
    }
}

```

Now that you have defined your `DialogFragment`, you can use it within your Activity much as you would any fragment—by using the `FragmentManager`. The following Activity class, called `SimpleFragDialogActivity`, has a layout resource that contains two Button controls, each of which triggers a new instance of the `MyAlertDialogFragment` to be generated and shown. The `show()` method of the `DialogFragment` is used to display the dialog, adding the fragment to the `FragmentManager` and passing in a little bit of information to configure the specific instance of the `DialogFragment` and its internal `AlertDialog`.

```

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

        // Handle Alert Dialog Button
        Button launchAlertDialog = (Button) findViewById(
            R.id.Button_AlertDialog);
        launchAlertDialog.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                showDialogFragment("Fragment Instance One");
            }
        });

        // Handle Alert Dialog 2 Button
        Button launchAlertDialog2 = (Button) findViewById(
            R.id.Button_AlertDialog2);
        launchAlertDialog2.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                showDialogFragment("Fragment Instance Two");
            }
        });
    }
}

```

```
void showDialogFragment(String strFragmentNumber) {  
    DialogFragment newFragment = MyAlertDialogFragment  
        .newInstance(strFragmentNumber);  
    newFragment.show(getFragmentManager(), strFragmentNumber);  
}  
  
public void doPositiveClick(String strFragmentNumber) {  
    Toast.makeText(getApplicationContext(),  
        "Clicked OK! (" + strFragmentNumber + ")",  
        Toast.LENGTH_SHORT).show();  
}  
}
```

Figure 11.2 shows a `DialogFragment` being displayed to the user.

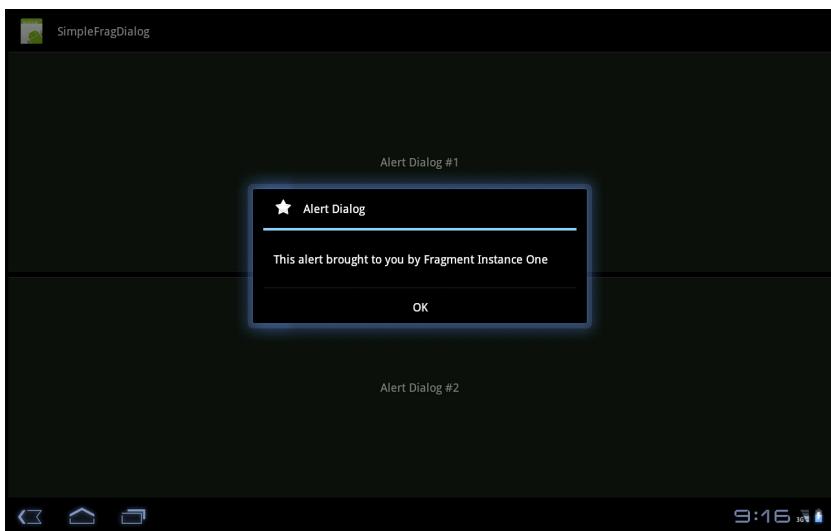


Figure 11.2 Using `DialogFragment` instances in an Activity.

We've shown you how to create `Dialog` controls that are managed by a `Fragment`, but when you start thinking of dialogs as fragments, you will see that this makes your dialogs much more powerful. For example, your `Dialog` controls can now be used by different activities because their lifecycle is managed inside the `Fragment`, not the `Activity`.

Also, `DialogFragment` instances can be traditional pop-ups (as shown in the example provided) or they can be embedded like any other `Fragment`. Why might you want to embed a `Dialog`? Consider the following example: You've created a picture gallery application and implemented a custom dialog that displays a larger-sized image when

you click a thumbnail. On small-screened devices, you might want this to be a pop-up `Dialog`, but on a tablet or TV, you might have the screen space to show the larger-sized graphic off to the right or below the thumbnails. This would be a good opportunity to take advantage of code reuse and simply embed your `Dialog`.

## Summary

Dialogs are useful controls for keeping your Android application user interfaces clean and user friendly. Many types of `Dialog` controls are defined in the Android SDK, and you can create custom `Dialog` controls if none of the canned controls suit your purposes.

Developers should be aware that there are two different approaches to implementing `Dialog` controls within applications. With the legacy method, the `Activity` class manages its `Dialog` controls using a number of straightforward callbacks. This method is backward-compatible without any issues, but it does not work well with the new Fragment-oriented user interface design paradigms used in Honeycomb and beyond. The second method involves using a special type of `Fragment` called a `DialogFragment`. This method decouples your dialogs from the `Activity` and instead treats them like fragments. They can also be used by other `Fragment` instances as well.

## References and More Information

Android SDK Reference regarding the application `Dialog` class:

<http://d.android.com/reference/android/app/Dialog.html>

Android SDK Reference regarding the application `AlertDialog` class:

<http://d.android.com/reference/android/app/AlertDialog.html>

Android SDK Reference regarding the application `DatePickerDialog` class:

<http://d.android.com/reference/android/app/DatePickerDialog.html>

Android SDK Reference regarding the application `TimePickerDialog` class:

<http://d.android.com/reference/android/app/TimePickerDialog.html>

Android SDK Reference regarding the application `ProgressDialog` class:

<http://d.android.com/reference/android/app/ProgressDialog.html>

Android SDK Reference regarding the application `CharacterPickerDialog` class:

<http://d.android.com/reference/android/text/method/CharacterPickerDialog.html>

Android SDK Reference regarding the application `DialogFragment` class:

<http://d.android.com/reference/android/app/DialogFragment.html>

Android Dev Guide: “Creating Dialogs”:

<http://d.android.com/guide/topics/ui/dialogs.html>

Android `DialogFragment` Reference: “Selecting Between Dialog and Embedding”:

<http://d.android.com/reference/android/app/DialogFragment.html#DialogOrEmbed>

# IV

## Android Application Design Essentials

- 12** Using Android Preferences
- 13** Working with Files and Directories
- 14** Using Content Providers
- 15** Designing Compatible Applications

*This page intentionally left blank*

# Using Android Preferences

Applications are about functionality and data. In this chapter, we explore the simplest way you can store, manage, and share application data persistently within your Android applications: by using shared preferences. The Android SDK includes a number of helpful APIs for storing and retrieving application preferences in different ways. Preferences are stored as groups of key/value pairs that can be used by the application. Shared preferences are most appropriate for storing simple kinds of data, such as application state and user settings, in a persistent fashion.

## Working with Application Preferences

Many applications need a lightweight data-storage mechanism called shared preferences for storing application state, simple user information, configuration options, and other such information. The Android SDK provides a simple preferences system for storing primitive application data at the `Activity` level and preferences shared across all of an application's activities.



### Tip

Many of the code examples provided in this section are taken from the `SimplePreferences` application. The source code for the `SimplePreferences` application is provided for download on the book's websites.

## Determining When Preferences Are Appropriate

Application preferences are sets of data values that are stored *persistently*, meaning that the preference data persists across application lifecycle events. In other words, the application or device can be started and stopped, turned on and off, without losing the data.

Many simple data values can be stored as application preferences. For example, your application might want to store the username of the application user. The application could use a single preference to store this information:

- The data type of the preference is a `String`.
- The key for the stored value is a `String` called “`UserName`”.
- The value for the data is the username “`HarperLee1926`”.

## Storing Different Types of Preference Values

Preferences are stored as groups of key/value pairs. The following data types are supported as preference setting values:

- Boolean values
- Float values
- Integer values
- Long values
- String values
- A set of multiple String values (new as of API Level 11)

Preference functionality can be found in the `SharedPreferences` interface of the `android.content` package. To add preferences support to your application, you must take the following steps:

1. Retrieve an instance of a `SharedPreferences` object.
2. Create a `SharedPreferences.Editor` to modify the preference content.
3. Make changes to the preferences using the Editor.
4. Commit your changes.

## Creating Private Preferences for Use by a Single Activity

Individual activities can have their own private preferences, though still represented by the `SharedPreferences` class. These preferences are for the specific `Activity` only and are not shared with other activities within the application. The activity gets only one group of private preferences, which are simply named after the `Activity` class. The following code retrieves an `Activity` class’s private preferences, called from within the `Activity`:

```
import android.content.SharedPreferences;  
...  
SharedPreferences settingsActivity = getPreferences(MODE_PRIVATE);
```

You have now retrieved the private preferences for that specific `Activity` class. Because the underlying name is based on the `Activity` class, any change to the `Activity` class will change what preferences are read.

## Creating Shared Preferences for Use by Multiple Activities

Creating shared preferences is similar. The only two differences are that we must name our preference set and use a different call to get the preference instance:

```
import android.content.SharedPreferences;  
...  
SharedPreferences settings =  
    getSharedPreferences("MyCustomSharedPreferences", MODE_PRIVATE);
```

You have now retrieved the shared preferences for the application. You can access these shared preferences by name from any activity in the application. There is no limit to the number of different shared preferences you can create. For example, you could have some shared preferences called “UserNetworkPreferences” and another called “AppDisplayPreferences”. How you organize shared preferences is up to you. However, you should declare the name of your preferences as a variable so that you can reuse the name across multiple activities consistently. Here’s an example:

```
public static final String PREFERENCE_FILENAME = "AppPrefs";
```

## Searching and Reading Preferences

Reading preferences is straightforward. Simply retrieve the `SharedPreferences` instance you want to read. You can check for a preference by name, retrieve strongly typed preferences, and register to listen for changes to the preferences. Table 12.1 describes some helpful methods in the `SharedPreferences` interface.

Table 12.1 Important `android.content.SharedPreferences` Methods

Method	Purpose
<code>SharedPreferences.contains()</code>	Sees whether a specific preference exists by name.
<code>SharedPreferences.edit()</code>	Retrieves the editor to change these preferences.
<code>SharedPreferences.getAll()</code>	Retrieves a map of all preference key/value pairs.
<code>SharedPreferences.getBoolean()</code>	Retrieves a specific Boolean-type preference by name.
<code>SharedPreferences.getFloat()</code>	Retrieves a specific Float-type preference by name.
<code>SharedPreferences.getInt()</code>	Retrieves a specific Integer-type preference by name.
<code>SharedPreferences.getLong()</code>	Retrieves a specific Long-type preference by name.

Table 12.1 **Continued**

Method	Purpose
<code>SharedPreferences.getString()</code>	Retrieves a specific String-type preference by name.
<code>SharedPreferences.getStringSet()</code>	Retrieves a specific set of String preferences by name. (Method added in API Level 11.)

## Adding, Updating, and Deleting Preferences

To change preferences, you need to open the preference Editor, make your changes, and commit them. Table 12.2 describes some helpful methods in the `SharedPreferences.Editor` interface.

Table 12.2 **Important android.content.SharedPreferences.Editor Methods**

Method	Purpose
<code>SharedPreferences.Editor.clear()</code>	Removes all preferences. This operation happens before any put operation, regardless of when it is called within an editing session; then all other changes are made and committed.
<code>SharedPreferences.Editor.remove()</code>	Removes a specific preference by name. This operation happens before any put operation, regardless of when it is called within an editing session; then all other changes are made and committed.
<code>SharedPreferences.Editor.putBoolean()</code>	Sets a specific Boolean-type preference by name.
<code>SharedPreferences.Editor.putFloat()</code>	Sets a specific Float-type preference by name.
<code>SharedPreferences.Editor.putInt()</code>	Sets a specific Integer-type preference by name.
<code>SharedPreferences.Editor.putLong()</code>	Sets a specific Long-type preference by name.
<code>SharedPreferences.Editor.putString()</code>	Sets a specific String-type preference by name.
<code>SharedPreferences.Editor.putStringSet()</code>	Sets a specific set of String-type preferences by name. (Method added in API Level 11.)
<code>SharedPreferences.Editor.commit()</code>	Commits all changes from this editing session.

Table 12.2 **Continued**

Method	Purpose
<code>SharedPreferences.Editor.apply()</code>	Much like the <code>commit()</code> method, this method commits all preference changes from this editing session. However, this method commits the changes to in-memory <code>SharedPreferences</code> immediately, but commits the changes to disk asynchronously within the application lifecycle. (Method added in API Level 9.)

The following block of code retrieves an `Activity` class's private preferences, opens the preference editor, adds a long-type preference called `SomeLong`, and saves the change:

```
import android.content.SharedPreferences;
...
SharedPreferences settingsActivity = getPreferences(MODE_PRIVATE);
SharedPreferences.Editor prefEditor = settingsActivity.edit();
prefEditor.putLong("SomeLong", java.lang.Long.MIN_VALUE);
prefEditor.commit();
```

Note that if you're targeting devices that run at least API Level 9 (Android 2.3 and higher), you would benefit from using the `apply()` method instead of the `commit()` method in the preceding code. However, if you need to support legacy versions of Android, which make up a good portion of the market, you'll want to stick with the `commit()` method, or check at runtime before calling the most appropriate method. Even when you are writing as little as one preference, using `apply` could smooth out the operation because any call to the file system may block for a noticeable (and therefore unacceptable) length of time.

## Reacting to Preference Changes

Your application can listen for, and react to, changes to shared preferences by implementing a listener and registering it with the specific `SharedPreferences` object using the `registerOnSharedPreferenceChangeListener()` and `unregisterOnSharedPreferenceChangeListener()` methods. This interface class has just one callback, which passes your code the shared preferences object that changed and which specific preference key name changed.

## Finding Preferences Data on the Android File System

Internally, application preferences are stored as XML files. You can access the preferences file using the File Explorer via DDMS. You find these files on the Android file system in the following directory:

`/data/data/<package name>/shared_prefs/<preferences filename>.xml`

The preferences filename is the `Activity` class name for private preferences or the specific name you give for the shared preferences. Here is an example of the XML file contents of a preference file with some simple values:

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<map>
    <string name="String_Pref">Test String</string>
    <int name="Int_Pref" value="-2147483648" />
    <float name="Float_Pref" value="-Infinity" />
    <long name="Long_Pref" value="9223372036854775807" />
    <boolean name="Boolean_Pref" value="false" />
</map>
```

Understanding the application preferences file format can be helpful for testing purposes. You can use Dalvik Debug Monitor Service (DDMS) to copy the preferences files to and from the device. Since the shared preferences are just a file, regular file permissions apply. When creating the file, you specify the mode (permissions) for the file. This determines if the file is readable outside of the existing package.

#### Note

For more information about using DDMS and the File Explorer, please see Appendix B, “The Android DDMS Quick-Start Guide.”

## Creating Manageable User Preferences

You now understand how to store and retrieve shared preferences programmatically. This works very well for keeping application state and such, but what if you have a set of user settings and you want to create a simple, consistent, and platform-standard way in which the user can edit them. Good news! You can use the handy `PreferenceActivity` class (`android.preference.PreferenceActivity`) to easily achieve this goal.

#### Tip

Many of the code examples provided in this section are taken from the `UserPrefs` application. The source code for the `UserPrefs` application is provided for download on the book’s websites.

Implementing a `PreferenceActivity`-based solution requires the following steps:

1. Define the preference set in a preference resource file.
2. Implement a `PreferenceActivity` class and tie it to the preference resource file.
3. Hook up the activity within your application as you normally would. For example, register it in the manifest file, start the activity as normal, and so on.

Now let’s look at these steps in more detail.

## Creating a Preference Resource File

First, you create an XML resource file to define the preferences your users are allowed to edit. A preference resource file contains a root level `<PreferenceScreen>` tag, followed by various preference types. These preference types are based on the `Preference` class (`android.preference.Preference`) and its subclasses, such as `CheckBoxPreference`, `EditTextPreference`, `ListPreference`, `MultiSelectListPreference`, and more. Some preferences have been around since the Android SDK was first released, whereas others, such as the `MultiSelectListPreference` class, have been introduced only recently and are not backward compatible with older devices.

Each preference should have some metadata, such as a title and some summary text, that will be displayed to the user. You can also specify default values, and for those preferences that launch dialogs, the dialog prompt. For the specific metadata associated with a given preference type, see its subclass attributes in the Android SDK documentation. Here are some common `Preference` attributes that most preferences should set:

- The `android:key` attribute is used to specify the key name for the shared preference.
- The `android:title` attribute is used to specify the friendly name of the preference, as shown on the editing screen.
- The `android:summary` attribute is used to give more details about the preference, as shown on the editing screen.
- The `android.defaultValue` attribute is used to specify a default value of the preference.

Like any resource files, preference resource files can use raw strings, or reference string resources. The following preference resource file example does a bit of both (the string array resources are defined elsewhere in the `strings.xml` resource file):

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
    xmlns:android="http://schemas.android.com/apk/res/android">
    <EditTextPreference
        android:key="username"
        android:title="Username"
        android:summary="This is your ACME Service username"
        android:defaultValue=""
        android:dialogTitle="Enter your ACME Service username:" />
    <PreferenceCategory
        android:title="Game Settings">
        <CheckBoxPreference
            android:key="bSoundOn"
            android:title="Enable Sound"
            android:summary="Turn sound on and off in the game"
            android:defaultValue="true" />
```

```

<CheckBoxPreference
    android:key="bAllowCheats"
    android:title="Enable Cheating"
    android:summary="Turn the ability to cheat on and off in the game"
    android:defaultValue="false" />
</PreferenceCategory>
<PreferenceCategory
    android:title="Game Character Settings">
    <ListPreference
        android:key="gender"
        android:title="Game Character Gender"
        android:summary="This is the gender of your game character"
        android:entries="@array/char_gender_types"
        android:entryValues="@array/char_genders"
        android:dialogTitle="Choose a gender for your character:" />
    <ListPreference
        android:key="race"
        android:title="Game Character Race"
        android:summary="This is the race of your game character"
        android:entries="@array/char_race_types"
        android:entryValues="@array/char_races"
        android:dialogTitle="Choose a race for your character:" />
</PreferenceCategory>
</PreferenceScreen>

```

This XML preference file is organized into two categories and defines fields for collecting several pieces of information, including a username (`String`), sound setting (`boolean`), cheat setting (`boolean`), character gender (fixed `String`), and character race (fixed `String`).

For instance, this example uses the `CheckBoxPreference` type to manage boolean shared preference values, such as game settings like whether or not sound is enabled or whether cheating is allowed. Boolean values are checked on and off straight from the screen. It uses the `EditTextPreference` type to manage the username, and it uses `ListPreference` types to allow the user to choose from a list of options. Finally, the settings are organized into categories using `<PreferenceCategory>` tags.

Next, you need to wire up your `PreferenceActivity` class and tell it about your preference resource file.

## Using the `PreferenceActivity` Class

The `PreferenceActivity` class (`android.preference.PreferenceActivity`) is a helper class that can load up your XML preferences resource file and transform it into a standard settings screen, much like you see in the Android device settings. Figure 12.1 shows what the screen for the preference resource file discussed in the previous section looks like when loaded into a `PreferenceActivity` class.

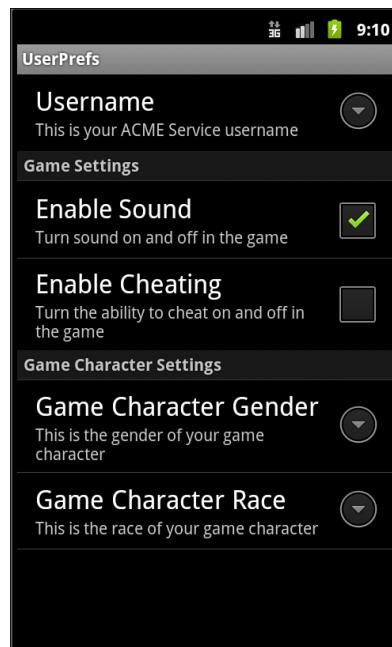


Figure 12.1 Game settings managed with `PreferenceActivity`.

To wire up your new preference resource file, create a new class that extends the `PreferencesActivity` class within your application. Next, override the `onCreate()` method of your class. Tie the preference resource file to the class using the `addPreferencesFromResource()` method. You will also want to retrieve an instance of the `PreferenceManager` (`android.preference.PreferenceManager`) and set the name of these preferences for use in the rest of your application at this time, if you're using a name other than the default. Here is the complete implementation of the `UserPrefsActivity` class, which encapsulates these steps:

```
import android.os.Bundle;
import android.preference.PreferenceActivity;
import android.preference.PreferenceManager;

public class UserPrefsActivity extends PreferenceActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        PreferenceManager manager = getPreferenceManager();
        manager.setSharedPreferencesName("user_prefs");
        addPreferencesFromResource(R.xml.userprefs);

    }
}
```

Now you can simply wire up the activity as you normally would. Don't forget to register it within your application's Android manifest file. When you run the application and start the `UserPrefsActivity`, you should see a screen that looks like Figure 12.1.

Trying to edit all other preferences will launch a dialog with the appropriate type of prompt (`EditText` or `Spinner` control), as shown in Figures 12.2 and 12.3.

Use the `EditTextPreference` type to manage string shared preference values, such as usernames, as shown in Figure 12.2.

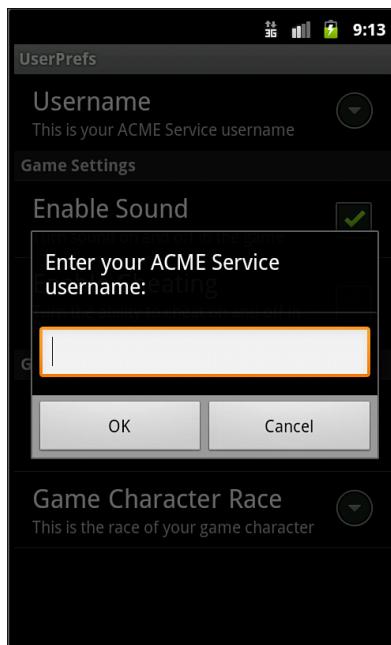


Figure 12.2 Editing an `EditText (String)` preference.

Use the `ListPreference` type to force the user to choose from a list of options, as shown in Figure 12.3.



### Tip

We have shown the method for using the `PreferenceActivity` class that is compatible with all versions of the Android platform. However, if you are targeting devices running API Level 11 (Android 3.0) and later only, you'll want to use the fragment-based method calls associated with the newly redesigned `PreferenceActivity` class, as discussed in Chapter 10, "Working with Fragments." You'll find a full example of how to use these methods (which involve loading headers instead of the entire resource file) in the `PreferenceActivity` class documentation.

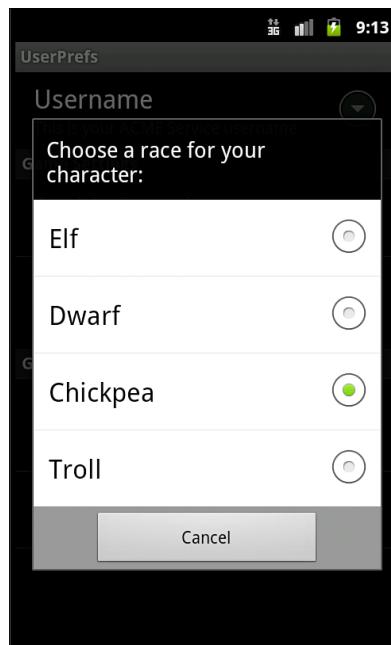


Figure 12.3 Editing a `ListPreference` (`String array`) preference.

## Summary

A variety of different ways to store and manage application data are available on the Android platform. The method you use depends on what kind of data you need to store. With these skills, you are well on your way to leveraging one of the more powerful and unique features of Android. Use shared preferences to store simple application data, such as strings and numbers, in a persistent manner. You can also use the `PreferenceActivity` class to simplify the creation of user preference screens within your application that use the standard look and feel of the platform your application is running on.

## References and More Information

Android SDK Reference regarding the `SharedPreferences` interface:

<http://d.android.com/reference/android/content/SharedPreferences.html>

Android SDK Reference regarding the `SharedPreferences.Editor` interface:

<http://d.android.com/reference/android/content/SharedPreferences.Editor.html>

Android SDK Reference regarding the `PreferenceActivity` class:

<http://d.android.com/reference/android/preference/PreferenceActivity.html>

Android SDK Reference regarding the `PreferenceScreen` class:  
<http://d.android.com/reference/android/preference/PreferenceScreen.html>

Android SDK Reference regarding the `PreferenceCategory` class:  
<http://d.android.com/reference/android/preference/PreferenceCategory.html>

Android SDK Reference regarding the `Preference` class:  
<http://d.android.com/reference/android/preference/Preference.html>

Android SDK Reference regarding the `CheckBoxPreference` class:  
<http://d.android.com/reference/android/preference/CheckBoxPreference.html>

Android SDK Reference regarding the `EditTextPreference` class:  
<http://d.android.com/reference/android/preference/EditTextPreference.html>

Android SDK Reference regarding the `ListPreference` class:  
<http://d.android.com/reference/android/preference/ListPreference.html>

# Working with Files and Directories

Android applications can store raw files on the device using a variety of methods. The Android SDK includes a number of helpful APIs for working with private application and cache files as well as accessing external files on removable storage such as SD cards. Developers who need to store information safely and persistently will find the available file management APIs familiar and easy to use.

## Working with Application Data on the Device

As discussed in the previous chapter, shared preferences provide a simple mechanism for storing simple application data persistently. However, many applications require a more robust solution that allows for any type of data to be stored and accessed in a persistent fashion. Some types of data that an application might want to store include the following:

- **Multimedia content such as images, sounds, video, and other complex information:** These types of data structures are not supported as shared preferences. You may, however, store a shared preference that includes the file path or URI to the multimedia, and store the multimedia on the device file system or download it just when needed.
- **Content downloaded from a network:** As mobile devices, Android devices are not guaranteed to have persistent network connections. Ideally, an application will download content from the network once, and keep it as long as necessary. Sometimes contents should be kept indefinitely, whereas other circumstances simply require contents to be cached for a time.
- **Complex content generated by the application:** Android devices function under more strict memory and storage constraints than desktop computers and servers do. Therefore, if your application has taken a long time to process data and come up with a result, that result should be stored for reuse, as opposed to re-creating it on demand.

Android applications can create and use directories and files to store their data in a variety of ways. The most common ways include the following:

- Storing private application data under the application directory.
- Caching data under the application’s cache directory.
- Storing shared application data on external storage devices or shared device directory areas.



#### Note

You can use Dalvik Debug Monitor Service (DDMS) to copy files to and from the device. For more information about using DDMS and the File Explorer, please see Appendix B, “The Android DDMS Quick-Start Guide.”

## Practicing Good File Management

You should follow a number of best practices when working with files on the Android file system. Here are a few of the most important ones:

- Any time you read or write data to disk, you are performing intensive, blocking operations and using valuable device resources. Therefore, in most cases, file-access functionality of your applications should not be performed on the main UI thread of the application. Instead, these operations should be handled asynchronously using threads, `AsyncTask` objects, or other asynchronous methods. Even working with small files can slow down the UI thread due to the nature of the underlying file system and hardware.
- Android devices have limited storage capacity. Therefore, store only what you need to store, and clean up old data when it is no longer needed to free up space on the device. Use external storage whenever it is appropriate to give the user more flexibility.
- Be a good “citizen” on the device: Be sure to check for availability of resources such as disk space and external storage opportunities prior to using them and causing errors or crashes. Also, don’t forget to set appropriate file permissions for new files, and release resources when you’re not using them (in other words, if you open them, close them, and so on).
- Implement efficient file-access algorithms for reading, writing, and parsing file contents. Use the many profiling tools available as part of the Android SDK to identify and improve the performance of your code. A good place to start is with the `StrictMode` API (`android.os.StrictMode`).
- If the data the application needs to store is well structured, you may want to consider using a SQLite database to store your application data. We discuss application databases in *Android Wireless Application Development Volume II: Advanced Topics*.

- Test your application on real devices. Different devices have different speed processors. Do not assume that because your application runs smoothly on the emulator it will run as such on real devices. If you’re using external storage, test when external storage is not available.

Let’s explore at how file management is achieved on the Android platform.

## Understanding Android File Permissions

Remember from Chapter 1, “Introducing Android,” that each Android application is its own user on the underlying Linux operating system. It has its own application directory and files. Files created in the application’s directory are private to that application by default.

Files can be created on the Android file system with different permissions. These permissions specify who can access the file. Three permission modes are most commonly used when creating files. These permission modes are defined in the `Context` class (`android.content.Context`):

- `MODE_PRIVATE` (the default) is used to create a file that can only be accessed by the “owner” application itself. From a Linux perspective, this means the specific user identifier. The value of `MODE_PRIVATE` is 0, so you may see this used in legacy code.
- `MODE_WORLD_READABLE` is used to create a file that can be read by all other applications, or user identifiers, on the device file system. However, the file can only be altered by the “owner” application.
- `MODE_WORLD_WRITEABLE` is used to create a file that can be modified, or written to, by all other applications, or user identifiers, on the device file system.

### Tip

If your application wants to create a file and grant other applications both read and write access, you will want to OR the appropriate modes together, like this: `MODE_WORLD_READABLE | MODE_WORLD_WRITEABLE`.

An application does not need any special Android manifest file permissions to access its own private file system area. However, if your application wants to access external storage, it will need to register for the `WRITE_EXTERNAL_STORAGE` permission.

## Working with Files and Directories

Within the Android SDK, you can also find a variety of standard Java file utility classes (such as `java.io`) for handling different types of files, such as text files, binary files, and XML files. In Chapter 7, “Managing Application Resources,” you learned that Android

applications can also include raw and XML files as resources. Retrieving the file handle to a resource file is performed slightly differently than accessing files on the device file system, but once you have a file handle, either method allows you to perform read operations and the like in the same fashion. After all, a file is a file.

Clearly Android application file resources are part of the application package and are therefore only accessible to the application itself. But what about file system files?

Android application files are stored in a standard directory hierarchy on the Android file system.

Generally speaking, applications access the Android device file system using methods within the `Context` class (`android.content.Context`). The application, or any `Activity` class, can use the application `Context` to access its private application file directory or cache directory. From here, you can add, remove, and access files associated with your application. By default, these files are private to the application and cannot be accessed by other applications or by the user.



### Tip

Many of the code examples provided in this section are taken from the `SimpleFiles` and `FileStreamOfConsciousness` applications. The `SimpleFiles` application demonstrates basic file and directory operations; it has no user interface (just LogCat output). The `FileStreamOfConsciousness` application demonstrates how to log strings to a file as a chat stream; this application is multithreaded. The source code for these applications is provided for download on the book's website.

## Exploring with the Android Application Directories

Android application data is stored on the Android file system in the following top-level directory:

```
/data/data/<package name>/
```

Several default subdirectories are created for storing databases, preferences, and files as necessary. The actual location of these directories varies by device. You can also create other custom directories as needed. File operations all begin by interacting with the application `Context` object. Table 13.1 lists some important methods available for application file management. You can use all the standard `java.io` package utilities to work with `FileStream` objects and such.

Table 13.1 Important `android.content.Context` File Management Methods

Method	Purpose
<code>Context.deleteFile()</code>	Delete a private application file by name. Note: You can also use the <code>File</code> class methods.
<code>Context fileList()</code>	Gets a list of all files in the <code>/files</code> subdirectory.

Table 13.1 Continued

Method	Purpose
Context.getCacheDir()	Retrieves the application /cache subdirectory.
Context.getDir()	Creates or retrieves an application subdirectory by name.
Context.getExternalCacheDir()	Retrieves the /cache subdirectory on the external filesystem (API Level 8).
Context.getExternalFilesDir()	Retrieves the /files subdirectory on the external filesystem (API Level 8).
Context.GetFilesDir()	Retrieves the application /files subdirectory.
Context.getFileStreamPath()	Returns the absolute file path to the application /files subdirectory.
Context.openFileInput()	Opens a private application file for reading.
Context.openFileOutput()	Opens a private application file for writing.

### Creating and Writing to Files to the Default Application Directory

Android applications that require only the occasional file to be created should rely upon the helpful Context class method called `openFileOutput()`. Use this method to create files in the default location under the application data directory:

```
/data/data/<package name>/files/
```

For example, the following code snippet creates and opens a file called `Filename.txt`. We write a single line of text to the file and then close the file:

```
import java.io.FileOutputStream;
...
FileOutputStream fos;
String strFileContents = "Some text to write to the file.";
fos = openFileOutput("Filename.txt", MODE_PRIVATE);
fos.write(strFileContents.getBytes());
fos.close();
```

We can append data to the file by opening it with the mode set to `MODE_APPEND`:

```
import java.io.FileOutputStream;
...
FileOutputStream fos;
String strFileContents = "More text to write to the file.";
fos = openFileOutput("Filename.txt", MODE_APPEND);
fos.write(strFileContents.getBytes());
fos.close();
```

The file we created has the following path on the Android file system:

```
/data/data/<package name>/files/Filename.txt
```

## Reading from Files in the Default Application Directory

Again we have a shortcut for reading files stored in the default /files subdirectory. The following code snippet opens a file called `Filename.txt` for read operations:

```
import java.io.FileInputStream;
...
String strFileName = "Filename.txt";
FileInputStream fis = openFileInput(strFileName);
```

## Reading Raw Files Byte-by-Byte

You handle file-reading and -writing operations using standard Java methods. Check out the subclasses of `java.io.InputStream` for reading bytes from different types of primitive file types. For example, `DataInputStream` is useful for reading one line at a time. Here's a simple example of how to read a text file, line by line, and store it in a `StringBuffer`:

```
FileInputStream fis = openFileInput(filename);
StringBuffer sBuffer = new StringBuffer();
DataInputStream dataIO = new DataInputStream(fis);
String strLine = null;

while ((strLine = dataIO.readLine()) != null) {
    sBuffer.append(strLine + "\n");
}

dataIO.close();
fis.close();
```

## Reading XML Files

The Android SDK includes several utilities for working with XML files, including SAX, an XML Pull Parser, and limited DOM, Level 2 Core support. Table 13.2 lists the packages helpful for XML parsing on the Android platform.

Table 13.2 Important XML Utilities

Package or Class	Purpose
<code>android.sax.*</code>	Framework to write standard SAX handlers.
<code>android.util.Xml</code>	XML utilities, including the <code>XMLPullParser</code> creator
<code>org.xml.sax.*</code>	Core SAX functionality. Project: <a href="http://www.saxproject.org/">www.saxproject.org/</a> .
<code>javax.xml.*</code>	SAX and limited DOM, Level 2 Core support.
<code>org.w3c.dom</code>	Interfaces for DOM, Level 2 Core.
<code>org.xmlpull.*</code>	<code>XmlPullParser</code> and <code>XMLSerializer</code> interfaces as well as a <code>SAX2</code> Driver class. Project: <a href="http://www.xmlpull.org/">www.xmlpull.org/</a> .

Your XML parsing implementation will depend on which parser you choose to use. Back in Chapter 7, we discussed including raw XML resource files in your application package. Here is a simple example of how to load an XML resource file and parse it using an `XmlPullParser`.

The XML resource file contents, as defined in the `/res/xml/my_pets.xml` file, are as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Our pet list -->
<pets>
    <pet type="Bunny" name="Bit"/>
    <pet type="Bunny" name="Nibble"/>
    <pet type="Bunny" name="Stack"/>
    <pet type="Bunny" name="Queue"/>
    <pet type="Bunny" name="Heap"/>
    <pet type="Bunny" name="Null"/>
    <pet type="Fish" name="Nigiri"/>
    <pet type="Fish" name="Sashimi II"/>
    <pet type="Lovebird" name="Kiwi"/>
</pets>
```

The following code illustrates how to parse the preceding XML using a special pull parser designed for XML resource files:

```
XmlResourceParser myPets = getResources().getXml(R.xml.my_pets);
int eventType = -1;
while (eventType != XmlResourceParser.END_DOCUMENT) {
    if(eventType == XmlResourceParser.START_DOCUMENT) {
        Log.d(DEBUG_TAG, "Document Start");
    } else if(eventType == XmlResourceParser.START_TAG) {

        String strName = myPets.getName();
        if(strName.equals("pet")) {
            Log.d(DEBUG_TAG, "Found a PET");
            Log.d(DEBUG_TAG,
                  "Name: "+myPets.getAttributeValue(null, "name"));
            Log.d(DEBUG_TAG,
                  "Species: "+myPets.
                  getAttributeValue(null, "type"));
        }
    }
    eventType = myPets.next();
}
Log.d(DEBUG_TAG, "Document End");
```

**Tip**

You can review the complete implementation of this parser in the ResourceRoundup project found in the Chapter 7 code directory.

## Working with Other Directories and Files on the Android File System

Using `Context.openFileOutput()` and `Context.openFileInput()` method calls are great if you have a few files and you want them stored in the application's private `/files` subdirectory, but if you have more sophisticated file-management needs, you need to set up your own directory structure. To do this, you must interact with the Android file system using the standard `java.io.File` class methods.

The following code retrieves the `File` object for the `/files` application subdirectory and retrieves a list of all filenames in that directory:

```
import java.io.File;  
...  
File pathForAppFiles = getFilesDir();  
String[] fileList = pathForAppFiles.list();
```

Here is a more generic method to create a file on the file system. This method works anywhere on the Android file system you have permission to access, not just the `/files` directory:

```
import java.io.File;  
import java.io.FileOutputStream;  
...  
File fileDir = getFilesDir();  
String strNewFileName = "myFile.dat";  
String strFileContents = "Some data for our file";  
  
File newFile = new File(fileDir, strNewFileName);  
newFile.createNewFile();  
  
FileOutputStream fo =  
    new FileOutputStream(newFile.getAbsolutePath());  
fo.write(strFileContents.getBytes());  
fo.close();
```

You can use `File` objects to manage files within a desired directory and create subdirectories. For example, you might want to store “track” files within “album” directories. Or perhaps you want to create a file in a directory other than the default. Let’s say you want to cache some data to speed up your application’s performance and how often it accesses the network. In this instance, you might want to create a cache file. There is also a special

application directory for storing cache files. Cache files are stored in the following location on the Android file system, retrievable with a call to the `getCacheDir()` method:

`/data/data/<package name>/cache/`

The external cache directory, found via a call to the `getExternalCacheDir()` method, is not treated the same in that files are not automatically removed from it.

### Warning

Applications are responsible for managing their own cache directory and keeping it to a reasonable size (1MB is commonly recommended). The system places no limit on the amount of files in a cache directory. The Android file system deletes cache files from the internal cache directory (`getCacheDir()`) as needed when internal storage space is low, or when the user uninstalls the application.

The following code gets a `File` object for the `/cache` application subdirectory, creates a new file in that specific directory, writes some data to the file, closes the file, and then deletes it:

```
File pathCacheDir = getCacheDir();
String strCacheFileName = "myCacheFile.cache";
String strFileContents = "Some data for our file";

File newCacheFile = new File(pathCacheDir, strCacheFileName);
newCacheFile.createNewFile();

FileOutputStream foCache =
    new FileOutputStream(newCacheFile.getAbsolutePath());
foCache.write(strFileContents.getBytes());
foCache.close();

newCacheFile.delete();
```

### Creating and Writing Files to External Storage

Applications should store large amounts of data on external storage (using the SD card) rather than limited internal storage. You can access external file storage, such as the SD card, from within your application as well. This is a little trickier than working within the confines of the application directory, as SD cards are removable, and so you need to check to see if the storage is mounted before use.

### Tip

You can monitor file and directory activity on the Android file system using the `FileObserver` class (`android.os.FileObserver`). You can monitor storage capacity using the `StatFs` class (`android.os.StatFs`).

You can access external storage on the device using the `Environment` class (`android.os.Environment`). Begin by using the `getExternalStorageState()` method to check the mount status of external storage. You can store private application files on external storage, or you can store public, shared files such as media. If you want to store private application files, use the `getExternalFilesDir()` method of the `Context` class because these files will be cleaned up if the application is uninstalled later. The external cache is accessed using the similar `getExternalCacheDir()` method. However, if you want to store shared files such as pictures, movies, music, ringtones, or podcasts on external storage, you can use the `getExternalStoragePublicDirectory()` method of the `Environment` class to get the top-level directory used to store a specific file type.



### Tip

Applications that use external storage are best tested in real hardware, as opposed to the emulator. You'll want to make sure you thoroughly test your application with various external storage states, including mounted, unmounted, and read-only modes. Each device may have different physical paths, so directory names should not be hard coded.

## Summary

A variety of different ways can be used to store and manage application data on the Android platform. The method you use depends on what kind of data you need to store. Applications have access to the underlying Android file system, where they can store their own private files, as well as limited access to the file system at large. It is important to follow best practices, such as performing disk operations asynchronously, when working on the Android file system, because mobile devices have limited storage and computing power.

## References and More Information

Android SDK Reference regarding the `java.io` package:

<http://d.android.com/reference/java/io/package-summary.html>

Android SDK Reference regarding the `Context` interface:

<http://d.android.com/reference/android/content/Context.html>

Android SDK Reference regarding the `File` class:

<http://d.android.com/reference/java/io/File.html>

Android SDK Reference regarding the `Environment` class:

<http://d.android.com/reference/android/os/Environment.html>

Android Dev Guide: “Using the Internal Storage”:

<http://d.android.com/guide/topics/data/data-storage.html#filesInternal>

Android Dev Guide: “Using the External Storage”:

<http://d.android.com/guide/topics/data/data-storage.html#filesExternal>

# Using Content Providers

Applications can access data within other applications on the Android system through content provider interfaces and expose internal application data to other applications by becoming a content provider. Content providers are the way applications can access user information, including contact data, images, audio and video on the device, and much more. In this chapter, we take a look at some of the content providers available on the Android platform and what you can do with them. You learn how to create your own content providers in *Android Wireless Application Development Volume II: Advanced Topics*.

## Warning

Always run content provider code on test devices, not your personal devices. It is very easy to accidentally wipe out all of the Contacts database, your browser bookmarks, or other types of data on your devices. Consider this fair warning, because we discuss operations such as how to query (generally safe) and modify (not so safe) various types of device data in this chapter.



## Exploring Android's Content Providers

Android devices ship with a number of built-in applications, many of which expose their data as content providers. Your application can access content provider data from a variety of sources. You can find the content providers included with Android in the package `android.provider`. Table 14.1 lists some useful content providers in this package.

Table 14.1 Useful Built-In Content Providers

Provider	Purpose
AlarmClock	Set alarms within the alarm clock application (API Level 9)
Browser	Browser history and bookmarks
CalendarContract	Calendar and event information (API Level 14)
CallLog	Sent and received calls

Table 14.1 **Continued**

Provider	Purpose
ContactsContract	Phone contact database or phonebook
MediaStore	Audio/visual data on the phone and external storage
SearchRecentSuggestions	Create search suggestions appropriate to the application
Settings	Systemwide device settings and preferences
UserDictionary	A dictionary of user-defined words for use with predictive text input
VoicemailContract	A single unified place for the user to manage voicemail content from different sources (API Level 14)

Now let's look at some of the most popular and official content providers in more detail.



### Tip

Many of the code examples provided in this chapter use the `Activity` class methods to perform managed queries using the `managedQuery()` method. This method is still very popular and works just fine, but is officially deprecated. If you are targeting Honeycomb devices and later, you'll want to investigate using the new loaders available in the latest versions of the Android SDK (as well as the Android Support Package) instead of having the `Activity` manage your `Cursor`. We discuss the use of `LoaderManager` (`android.app.LoaderManager`) and loaders in *Android Wireless Application Development, Volume II: Advanced Topics*.

Keep in mind that accessing a content provider has ramifications to application performance and responsiveness, much like accessing device storage, a database, or the network. Therefore, in commercial applications, you should perform content provider queries and operations asynchronously off the main UI thread regardless of how fast they may appear, due to the nature of the underlying file system and hardware.

## Using the `MediaStore` Content Provider

You can use the `MediaStore` content provider to access media on the phone and on external storage devices. The primary types of media you can access are audio, images, and video. You can access these different types of media through their respective content provider classes under `android.provider.MediaStore`.

Most of the `MediaStore` classes allow full interaction with the data. You can retrieve, add, and delete media files from the device. There is also a handful of helper classes that define the most common data columns that can be requested.

Table 14.2 lists some commonly used classes you can find under `android.provider.MediaStore`.

Table 14.2 Common MediaStore Classes

Class	Purpose
Audio.Albums	Manages audio files organized by the album
Audio.Artists	Manages audio files by the artist who created them
Audio.Genres	Manages audio files belonging to a particular genre
Audio.Media	Manages audio files on the device
Audio.Playlists	Manages audio files that are part of a particular playlist
Files	Listing of all media files (API Level 11)
Images.Media	Manages image files on the device
Images.Thumbnails	Retrieves thumbnails for the image files
Video.Media	Manages video files on the device
Video.Thumbnails	Retrieves thumbnails for the video files

**Tip**

Many of the code examples provided in this section are taken from the SimpleContentProvider application. The source code for the SimpleContentProvider application is provided for download on the book website.

The following code demonstrates how to request data from a content provider. A query is made to the `MediaStore` to retrieve the titles of all the audio files on the SD card of the handset and their respective durations. This code requires that you load some audio files onto the virtual SD card in the emulator.

```
String[] requestedColumns = {
    MediaStore.Audio.Media.TITLE,
    MediaStore.Audio.Media.DURATION
};

Cursor cur = managedQuery(
    MediaStore.Audio.Media.EXTERNAL_CONTENT_URI,
    requestedColumns, null, null, null);

Log.d(DEBUG_TAG, "Audio files: " + cur.getCount());
Log.d(DEBUG_TAG, "Columns: " + cur.getColumnCount());

int name = cur.getColumnIndex(MediaStore.Audio.Media.TITLE);
int length = cur.getColumnIndex(MediaStore.Audio.Media.DURATION);

cur.moveToFirst();
while (!cur.isAfterLast()) {
    Log.d(DEBUG_TAG, "Title" + cur.getString(name));
    Log.d(DEBUG_TAG, "Length: " +
```

```

        cur.getInt(length) / 1000 + " seconds");
        cur.moveToNext();
    }
}

```

The `MediaStore.Audio.Media` class has predefined strings for every data field (or column) exposed by the content provider. You can limit the audio file data fields requested as part of the query by defining a string array with the column names required. In this case, we limit the results to only the track title and the duration of each audio file.

We then use a `managedQuery()` method call. The first parameter is the predefined URI of the content provider you want to query. The second parameter is the list of columns to return (audio file titles and durations). The third and fourth parameters control any selection-filtering arguments, and the fifth parameter provides a sort method for the results. We leave these `null` because we want all audio files at this location. By using the `managedQuery()` method, we get a managed Cursor as a result. We then examine our Cursor for the results.

## Using the CallLog Content Provider

Android provides a content provider to access the call log on the handset via the class `android.provider.CallLog`. At first glance, the `CallLog` might not seem to be a useful provider for developers, but it has some nifty features. You can use the `CallLog` to filter recently dialed calls, received calls, and missed calls. The date and duration of each call is logged and tied back to the Contact application for caller identification purposes.

The `CallLog` is a useful content provider for customer relationship management (CRM) applications. The user can also tag specific phone numbers with custom labels within the Contact application.

To demonstrate how the `CallLog` content provider works, let's look at a hypothetical situation where we want to generate a report of all calls to a number with the custom label `HourlyClient123`. Android allows for custom labels on these numbers, which we leverage for this example:

```

String[] requestedColumns = {
    CallLog.Calls.CACHED_NUMBER_LABEL,
    CallLog.Calls.DURATION
};

Cursor calls = managedQuery(
    CallLog.Calls.CONTENT_URI, requestedColumns,
    CallLog.Calls.CACHED_NUMBER_LABEL
    + " = ?", new String[] { "HourlyClient123" } , null);

Log.d(DEBUG_TAG, "Call count: " + calls.getCount());

int durIdx = calls.getColumnIndex(CallLog.Calls.DURATION);
int totalDuration = 0;

```

```
calls.moveToFirst();
while (!calls.isAfterLast()) {
    Log.d(DEBUG_TAG, "Duration: " + calls.getInt(durIdx));
    totalDuration += calls.getInt(durIdx);
    calls.moveToNext();
}

Log.d(DEBUG_TAG, "HourlyClient123 Total Call Duration: " + totalDuration);
```

This code is similar to the code shown for the `MediaStore` audio files. Again, we start with listing our requested columns: the call label and the duration of the call. This time, however, we don't want to get every call in the log, only those with a label of `HourlyClient123`. To filter the results of the query to this specific label, it is necessary to specify the third and fourth parameters of the `managedQuery()` call. Together, these two parameters are equivalent to a database `WHERE` clause. The third parameter specifies the format of the `WHERE` clause with the column name, with selection parameters (shown as `?`) for each selection argument value. The fourth parameter, the `String` array, provides the values to substitute for each of the selection arguments (`?`) in order as you would do for a simple SQLite database query.

As before, the `Activity` manages the `Cursor` object lifecycle. We use the same method to iterate the records of the `Cursor` and add up all the call durations.

## Accessing Content Providers That Require Permissions

Your application needs a special permission to access the information provided by the `CallLog` content provider. You can declare the `uses-permission` tag using the Eclipse Wizard or you can add the following to your `AndroidManifest.xml` file:

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

Although it's a tad confusing, there is no `CallLog` provider permission. Instead, applications that access the `CallLog` use the `READ_CONTACTS` permission. Although the values are cached within this content provider, the data is similar to what you might find in the contacts provider.



### Tip

You can find all available permissions in the class `android.Manifest.permission`.

## Using the Browser Content Provider

Another useful, built-in content provider is the `Browser`. The `Browser` content provider exposes the user's browser site history and bookmarked websites. You access this content provider via the `android.provider.Browser` class. As with the `CallLog` class, you can

use the information provided by the `Browser` content provider to generate statistics and to provide cross-application functionality. You might use the `Browser` content provider to add a bookmark for your application support website.

In this example, we query the `Browser` content provider to find the top five most frequently visited bookmarked sites.

```
String[] requestedColumns = {
    Browser.BookmarkColumns.TITLE,
    Browser.BookmarkColumns.VISITS,
    Browser.BookmarkColumns.BOOKMARK
};

Cursor faves = managedQuery(Browser.BOOKMARKS_URI, requestedColumns,
    Browser.BookmarkColumns.BOOKMARK + "=1", null,
    Browser.BookmarkColumns.VISITS + " DESC LIMIT 5");

Log.d(DEBUG_TAG, "Bookmarks count: " + faves.getCount());

int titleIdx = faves.getColumnIndex(Browser.BookmarkColumns.TITLE);
int visitsIdx = faves.getColumnIndex(Browser.BookmarkColumns.VISITS);
int bmIdx = faves.getColumnIndex(Browser.BookmarkColumns.BOOKMARK);

faves.moveToFirst();

while (!faves.isAfterLast()) {
    Log.d("SimpleBookmarks", faves.getString(titleIdx) + " visited "
        + faves.getInt(visitsIdx) + " times : "
        + (faves.getInt(bmIdx) != 0 ? "true" : "false"));
    faves.moveToNext();
}
```

Again, the requested columns are defined, the query is made, and the cursor iterates through the results.

Note that the `managedQuery()` call has become substantially more complex. Let's take a look at the parameters to this method in more detail. The first parameter, `Browser.BOOKMARKS_URI`, is a URI for all browser history, not only the bookmarked items. The second parameter defines the requested columns for the query results. The third parameter specifies that the `bookmark` property must be true. This parameter is needed in order to filter within the query. Now the results are only browser history entries that have been bookmarked. The fourth parameter, selection arguments, is used only when replacement values are used. It is not used in this case, so the value is set to `null`. Lastly, the fifth parameter specifies an order for the results (most visited in descending order). Retrieving browser history information requires setting the `READ_HISTORY_BOOKMARKS` permission.



### Tip

Notice that we also tacked on a `LIMIT` statement to the fifth parameter of `managedQuery()`. Although this is not specifically documented, we've found limiting the query results in this way works well and might even improve application performance in some situations where the query results are lengthy. Keep in mind that if the internal implementation of a content provider verified that the last parameter was only a valid `ORDER BY` clause, this may not always work. We are also taking advantage of the fact that most content providers are backed by SQLite. This need not be the case.

## Using the `CalendarContract` Content Provider

Introduced officially in Android 4.0 (API Level 14), the `CalendarContract` provider allows you to manage and interact with the user's calendar data on the device. You can use this content provider to create one-time and recurring events in a user's calendar, set reminders, and more, provided the device user has appropriately configured calendar accounts (for example, Microsoft Exchange). In addition to the fully featured content provider, you can also quickly trigger a new event to be added to the user's calendar using an `Intent`, like this:

```
Intent calIntent = new Intent(Intent.ACTION_INSERT);
calIntent.setData(CalendarContract.Events.CONTENT_URI);
calIntent.putExtra(Events.TITLE, "My Winter Holiday Party");
calIntent.putExtra(Events.EVENT_LOCATION, "My Ski Cabin at Tahoe");
calIntent.putExtra(Events.DESCRIPTION, "Hot chocolate, eggnog and sledding.");
startActivity(calIntent);
```

Here we seed the calendar event title, location, and description using the appropriate intent `Extras`. These fields will be set in a form that displays for the user, who will then need to confirm the event in the calendar app. For more information on `CalendarContract` intent usage, see our online article “Android Essentials: Adding Events to the User's Calendar,” available at <http://goo.gl/ZDbH5>.

## Using the `UserDictionary` Content Provider

Another useful content provider is the `UserDictionary` provider. You can use this content provider for predictive text input on text fields and other user input mechanisms. Individual words stored in the dictionary are weighted by frequency and organized by locale. You can use the `addWord()` method within the `UserDictionary.Words` class to add words to the custom user dictionary.

## Using the `VoicemailContract` Content Provider

The `VoicemailContract` content provider was introduced in API Level 14. You can use this content provider to add new voicemail content to the shared provider so that all voicemail content is accessible in one place. Application permissions, such as the

`ADD_VOICEMAIL` permission, are necessary for accessing this provider. For more information, see the Android SDK documentation for the `VoicemailContract` class at <http://d.android.com/reference/android/provider/VoicemailContract.html>.

## Using the Settings Content Provider

Another useful content provider is the `Settings` provider. You can use this content provider to access the device settings and user preferences. Settings are organized much as they are in the `Settings` application—by category. You can find information about the `Settings` content provider in the `android.provider.Settings` class. If your application needs to modify system settings, you'll need to register the `WRITE_SETTINGS` or `WRITE_SECURE_SETTINGS` permissions in your application's Android manifest file.

## Using the Contacts Content Providers

The `Contacts` database is one of the most commonly used applications on the mobile phone. People always want phone numbers handy for calling friends, family, coworkers, and clients. Additionally, most phones show the identity of the caller based on the `Contacts` application, including nicknames, photos, or icons.

Android provides a built-in Contact application, and the contact data is exposed to other Android applications using the content provider interface. As an application developer, this means you can leverage the user's contact data within your application for a more robust user experience.

The content provider for accessing user contacts was originally called `Contacts`. Android 2.0 (API Level 5) introduced an enhanced contacts management content provider class to manage the data available from the user's contacts. This provider, called `ContactsContract`, includes a subclass called `ContactsContract.Contacts`. This is the preferred contacts content provider moving forward. However, because contacts are a commonly used feature across all Android platform versions, we will provide some examples using both the original `Contacts` content provider method for backward compatibility and the `ContactsContract` provider for those targeting newer device platform versions.

Regardless of which method you use, your application needs special permission to access the private user information provided by either `Contacts` content provider. You must declare a `uses-permission` tag using the permission `READ_CONTACTS` to read this information. If your application modifies the Contact database, you'll need the `WRITE_CONTACTS` permission as well.



### Tip

Some of the code examples provided in this section are taken from the `SimpleContacts` application. The source code for the `SimpleContacts` application is provided for download on the book's websites.

## Working with the Legacy Contacts Content Provider

First, we'll look at how you access the older contacts provider. This works due to Android's tradition of forward compatibility, but many of the classes and methods used are now deprecated in the latest Android SDK versions. Here we have a simple query of the Contacts database. This short example simply shows querying for a single contact:

```
Cursor oneContact = managedQuery( People.CONTENT_URI, null, null, null,
    People.NAME + " DESC LIMIT 1");

Log.d(DEBUG_TAG, "Count: " + oneContact.getCount());
```

We used `LIMIT` to retrieve one contact record. If you actually look at the returned columns of data, you find that there is little more than the contact name and some indexes. The data fields are not explicitly returned. Instead, the results include the values needed to build specific URIs to those pieces of data. We need to request the data for the contact using these indexes.

Specifically, we retrieve the primary email and primary phone number for this contact, as follows:

```
int nameIdx = oneContact.getColumnIndex(ContactMethods.People.NAME);
int emailIDIdx = oneContact
    .getColumnIndex(ContactMethods.People.PRIMARY_EMAIL_ID);

int phoneIDIdx = oneContact
    .getColumnIndex(ContactMethods.People.PRIMARY_PHONE_ID);

if (oneContact.getCount() == 1) {
    oneContact.moveToFirst();
    int emailID = oneContact.getInt(emailIDIdx);
    int phoneID = oneContact.getInt(phoneIDIdx);
}
```

Now that we have the column index values for the contact's name, primary email address, and primary phone number, we need to build the Uri objects associated with those pieces of information and query for the primary email and primary phone number:

```
Uri emailUri = ContentUris.withAppendedId(
    ContactMethods.CONTENT_URI,
    emailID);

Uri phoneUri = ContentUris.withAppendedId(
    Phones.CONTENT_URI, phoneID);

Cursor primaryEmail = managedQuery(emailUri,
    new String[] {
        ContactMethods.DATA
```

```

        },
        null, null, null);

Cursor primaryNumber = managedQuery(phoneUri,
    new String[] {
        Contacts.Phones.NUMBER
    },
    null, null, null);

```

After retrieving the appropriate column indexes for a contact's specific email and phone number, we call `ContentUris.withAppendedId()` to create the new Uri objects from existing ones and the identifiers we now have. This enables direct selection of a particular row from the table when the index of that row is known. You can use a selection parameter to do this as well. Lastly, we used the two new Uri objects to perform two calls to `managedQuery()`.

Now we take a shortcut with the requested columns `String` array because each query only has one column:

```

String name = oneContact.getString(nameIdx);
primaryEmail.moveToFirst();
String email = primaryEmail.getString(0);
primaryNumber.moveToFirst();
String number = primaryNumber.getString(0);

```

If an email or phone number doesn't exist, an exception called `android.database.CursorIndexOutOfBoundsException` is thrown. This can be caught, or you can check to see that a result was actually returned in the `Cursor` first.

### Querying for a Specific Contact

If you thought that seemed like quite a lot of coding to get a phone number, you're not alone. For getting a quick piece of data, there is a faster way. The following block of code demonstrates how we can get the phone number and name for one contact:

```

String[] requestedColumns = {
    Contacts.Phones.NAME,
    Contacts.Phones.NUMBER,
};

Cursor contacts = managedQuery(
    Contacts.Phones.CONTENT_URI,
    requestedColumns,
    null,
    null, People.NAME + " DESC LIMIT 1");

int recordCount = contacts.getCount();
Log.d(DEBUG_TAG, "Contacts count: "
    + recordCount);

```

```
if (recordCount > 0) {  
  
    int nameIdx = contacts  
        .getColumnIndex(Contacts.Phones.NAME);  
    int phoneIdx = contacts  
        .getColumnIndex(Contacts.Phones.NUMBER);  
  
    contacts.moveToFirst();  
    Log.d(DEBUG_TAG, "Name: " + contacts.getString(nameIdx));  
    Log.d(DEBUG_TAG, "Phone: " + contacts.getString(phoneIdx));  
}
```

This block of code should look somewhat familiar, yet it is a much shorter and more straightforward method to query for phone numbers by contact name. The `Contacts.Phones.CONTENT_URI` contains phone numbers, but it also happens to have the contact name. This is similar to the `CallLog` content provider.

## Working with the `ContactsContract` Content Provider

Now let's turn our attention to the more recent contacts content provider: `ContactsContract.Contacts`. This provider, introduced in API Level 5 (Android 2.0), provides a robust contact content provider that suits the more robust Contacts application that has evolved along with the Android platform. The basics for accessing the content provider are very similar.

### Tip

The `ContactsContract` content provider was further enhanced in Android 4.0 (Ice Cream Sandwich, API Level 14) to incorporate substantive social network features. Some of the new features include managing the device user's identity, favorite methods of communication with specific contacts, as well as a new `INVITE_CONTACT` intent type for applications to use to make contact connections. The device user's personal profile is accessible through the `ContactsContract.Profile` class (requires the `READ_PROFILE` application permission). The device user's preferred methods of communicating with specific contacts can be accessed through the new `ContactsContract.DataUsageFeedback` class. For more information, see the Android SDK documentation for the `android.provider.ContactsContract` class.

For example, the following code performs the same function as the previous code example using the legacy `Contacts` content provider, but uses the newer `ContactsContract` provider instead:

```
String[] requestedColumns = {  
    ContactsContract.Contacts.DISPLAY_NAME,  
    ContactsContract.CommonDataKinds.Phone.NUMBER,  
};
```

```

Cursor contacts = managedQuery(ContactsContract.Data.CONTENT_URI,
    requestedColumns, null, null,
    ContactsContract.Contacts.DISPLAY_NAME + " DESC limit 1");

int recordCount = contacts.getCount();
Log.d(DEBUG_TAG, "Contacts count: " + recordCount);

if (recordCount > 0) {
    int nameIdx = contacts
        .getColumnIndex(ContactsContract.Contacts.DISPLAY_NAME);
    int phoneIdx = contacts
        .getColumnIndex(ContactsContract.CommonDataKinds.Phone.NUMBER);

    contacts.moveToFirst();
    Log.d(DEBUG_TAG, "Name: " + contacts.getString(nameIdx));
    Log.d(DEBUG_TAG, "Phone: " + contacts.getString(phoneIdx));
}

```

There are two main differences between this code and the previous legacy code. First, you're using a query URI provided from the `ContactsContract` provider called `ContactsContract.Data.CONTENT_URI`. Second, you're requesting different column names. The column names of the `ContactsContract` provider are organized more thoroughly to allow for far more dynamic contact configurations. This can make your queries slightly more complex. Luckily, the `ContactsContract.CommonDataKinds` class has a number of frequently used columns defined together. Table 14.3 shows some of the commonly used classes that can help you work with the `ContactsContract` content provider.

Table 14.3 Commonly Used `ContactsContract` Data Column Classes

Class	Purpose
<code>ContactsContract.CommonDataKinds</code>	Defines a number of frequently used Contact columns such as email, nickname, phone, and photo.
<code>ContactsContract.Contacts</code>	Defines the consolidated data associated with a contact. Some aggregation may be performed.
<code>ContactsContract.Data</code>	Defines the raw data associated with a single contact.
<code>ContactsContract.PhoneLookup</code>	Defines the phone columns and can be used to quickly look up a phone number for caller identification purposes.
<code>ContactsContract.StatusUpdates</code>	Defines the social networking columns and can be used to check the instant messaging status of a contact.

For more information on the `ContactsContract` provider, see the Android SDK documentation: <http://d.android.com/reference/android/provider/ContactsContract.html>.

## Modifying Content Providers Data

Content providers are not only static sources of data. They can also be used to add, update, and delete data, if the content provider application has implemented this functionality. Your application must have the appropriate permissions (that is, `WRITE_CONTACTS` as opposed to `READ_CONTACTS` if you were using the `Contacts` content provider) to perform some of these actions. Let's return to the legacy `Contacts` content provider and give some examples of how to modify the `Contacts` database.

### Adding Records

Using the legacy `Contacts` content provider, we can, for example, add a new record to the `Contacts` database programmatically, as shown here:

```
ContentValues values = new ContentValues() ;  
  
values.put(Contacts.People.NAME, "Sample User") ;  
  
Uri uri = getContentResolver().insert(  
    Contacts.People.CONTENT_URI, values) ;  
  
Uri phoneUri = Uri.withAppendedPath(uri,  
    Contacts.People.Phones.CONTENT_DIRECTORY) ;  
  
values.clear() ;  
  
values.put(Contacts.Phones.NUMBER, "2125551212") ;  
values.put(Contacts.Phones.TYPE, Contacts.Phones.TYPE_WORK) ;  
  
getContentResolver().insert(phoneUri, values) ;  
  
values.clear() ;  
  
values.put(Contacts.Phones.NUMBER, "3135551212") ;  
values.put(Contacts.Phones.TYPE, Contacts.Phones.TYPE_MOBILE) ;  
  
getContentResolver().insert(phoneUri, values) ;
```

Here we use the `ContentValues` class to insert records into the `Contacts` database on the device. The first action we take is to provide a name for the `Contacts.People.NAME` column. We need to create the contact with a name before we can assign information, such as phone numbers. Think of this as creating a row in a table that provides a one-to-many relationship to a phone number table.

Next, we insert the data in the database found at the `Contacts.People.CONTENT_URI` path. We use a call to `getContentResolver()` to retrieve the `ContentResolver` associated with our `Activity`. The return value is the `Uri` of our new contact. We need to use it for adding phone numbers to our new contact. We then reuse the `ContentValues` instance by clearing it and adding a `Contacts.Phones.NUMBER` and the `Contacts.Phones.TYPE` for it. Using the `ContentResolver`, we insert this data into the newly created `Uri`.



### Tip

At this point, you might be wondering how the structure of the data can be determined. The best way is to thoroughly examine the documentation from the specific content provider with which you want to integrate your application.

## Updating Records

Inserting data isn't the only change you can make. You can update one or more rows as well. The following block of code shows how to update data within a content provider. In this case, we update a note field for a specific contact, using its unique identifier (`rowId`).

```
ContentValues values = new ContentValues();
values.put(People.NOTES, "This is my boss");
Uri updateUri = ContentUris.withAppendedId(People.CONTENT_URI, rowId);
int rows = getContentResolver().update(updateUri, values, null, null);
Log.d(debugTag, "Rows updated: " + rows);
```

Again, we use an instance of the `ContentValues` object to map the data field we want to update with the data value—in this case, the note field. This replaces any current note stored in the `NOTES` field currently stored with the contact. We then create the `Uri` for the specific contact we are updating. A simple call to the `update()` method of the `ContentResolver` class completes our change. We can then confirm that only one row was updated.



### Tip

You can use the filter values when updating rows. This enables you to make changes to values across many rows at the same time. The content provider must support this, though. We have found that the contacts provider blocks this on the `People` URL, thus preventing developers from making sweeping or global changes to contacts.

## Deleting Records

Now that you cluttered up your contacts application with sample user data, you might want to delete some of it. Deleting data is fairly straightforward.

## Deleting All Records

The following code deletes all rows at the given URI. Keep in mind that you should execute operations like this with extreme care.

```
int rows = getContentResolver().delete(People.CONTENT_URI, null, null);
Log.d(debugTag, "Rows: " + rows);
```

The `delete()` method deletes all rows at a given URI filtered by the selection parameters, which, in this case, includes all rows at the `People.CONTENT_URI` location (in other words, all contact entries).

## Deleting Specific Records

Often you want to select specific rows to delete by adding the unique identifier index to the end of the URI or remove rows matching a particular pattern.

For example, the following deletion matches all contact records with the name Sample User, which we used when we created sample contacts previously in the chapter.

```
int rows = getContentResolver().delete(People.CONTENT_URI,
    People.NAME + "=?",
    new String[] {"Sample User"});
Log.d(debugTag, "Rows: " + rows);
```

# Using Third-Party Content Providers

Any application can implement a content provider to share its information safely and securely with other applications on the device. Some applications use content providers only to share information internally—within their own brand, for example. Others publish the specifications for their providers so that any other applications that want to integrate with them can.

If you poke around in the Android source code, or run across a content provider you want to use, consider this: A number of other content providers are available on the Android platform, especially those used by some of the typically installed Google applications (Calendar, Messaging, and so on). Be aware, though, that using undocumented content providers, simply because you happen to know how they work or have reverse-engineered them, is generally not a good idea. Use of undocumented and non-public content providers can make your application unstable. This post on the Android Developers blog makes a good case for why this sort of hacking should be discouraged in commercial applications: <http://android-developers.blogspot.com/2010/05/be-careful-with-content-providers.html>.

## Summary

Your application can leverage the data available within other Android applications, if they expose that data as a content provider. Content providers such as `MediaStore`, `Browser`, `CallLog`, and `Contacts` can be leveraged by other Android applications, resulting in a robust, immersive experience for users. Applications can also share data among themselves by becoming content providers. Becoming a content provider involves implementing a set of methods that manage how and what data you expose for use in other applications—a topic we cover in detail in *Android Wireless Application Development Volume II: Advanced Topics*.

## References and More Information

Android SDK Reference regarding the `android.provider` package:

<http://d.android.com/reference/android/provider/package-summary.html>

Android SDK Reference regarding the `AlarmClock` content provider:

<http://d.android.com/reference/android/provider/AlarmClock.html>

Android SDK Reference regarding the `Browser` content provider:

<http://d.android.com/reference/android/provider/Browser.html>

Android SDK Reference regarding the `CallLog` content provider:

<http://d.android.com/reference/android/provider/CallLog.html>

Android SDK Reference regarding the `Contacts` content provider:

<http://d.android.com/reference/android/provider/Contacts.html>

Android SDK Reference regarding the `ContactsContract` content provider:

<http://d.android.com/reference/android/provider/ContactsContract.html>

Android SDK Reference regarding the `MediaStore` content provider:

<http://d.android.com/reference/android/provider/MediaStore.html>

Android SDK Reference regarding the `Settings` content provider:

<http://d.android.com/reference/android/provider/Settings.html>

Android SDK Reference regarding the `SearchRecentSuggestions` content provider:

<http://d.android.com/reference/android/provider/SearchRecentSuggestions.html>

Android SDK Reference regarding the `UserDictionary` content provider:

<http://d.android.com/reference/android/provider/UserDictionary.html>

Android Dev Guide: “Content Providers”:

<http://d.android.com/guide/topics/providers/content-providers.html>

Android Dev Guide: “Using the Contacts API”:

<http://d.android.com/resources/articles/contacts.html>

# Designing Compatible Applications

There are now hundreds of different Android devices on the market worldwide—from smartphones to tablets and televisions. In this chapter, you learn how to design and develop Android applications that are compatible with a variety of devices despite differences in screen size, hardware, or platform version. We offer numerous tips for designing and developing your application to be compatible with many different devices. Finally, you learn how to internationalize your applications for foreign markets.

## Maximizing Application Compatibility

With dozens of manufacturers developing Android devices, we've seen an explosion of different device models—each with its own market differentiators and unique characteristics. Users now have choices, but these choices come at a cost. This proliferation of devices has led to what some developers call *fragmentation* and others call *compatibility issues*. Terminology aside, it has become a challenging task to develop Android applications that support a broad range of devices. Developers must contend with devices that support different platform versions (see Figure 15.1), hardware configurations (including optional hardware features) such as OpenGL versions (see Figure 15.2), and variations in screen sizes and densities (see Figure 15.3). The list of device differentiators is lengthy, and grows with each new device.

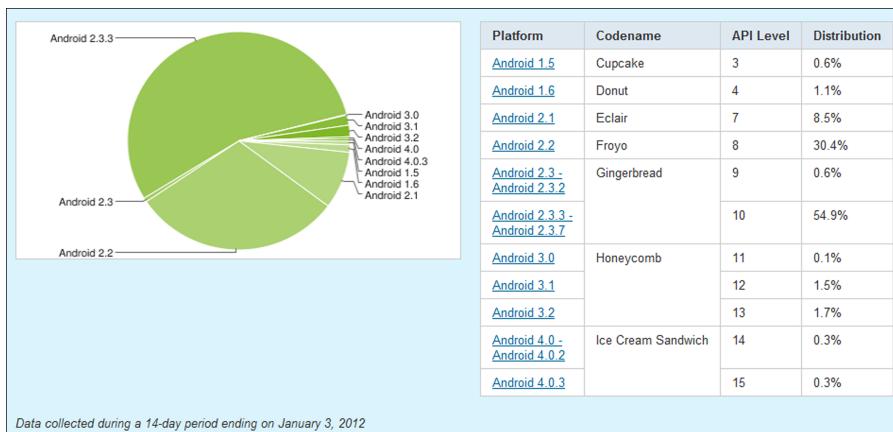


Figure 15.1 Android device statistics regarding platform version  
(source: <http://goo.gl/7HxNH>).

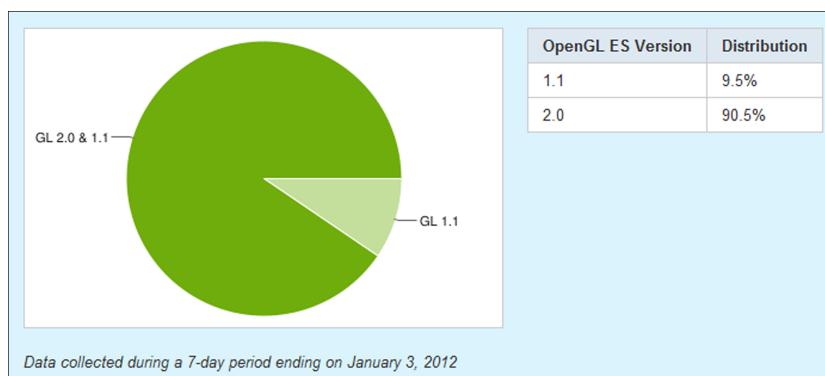


Figure 15.2 Android device statistics regarding OpenGL versions  
(source: <http://goo.gl/ngt8W>).

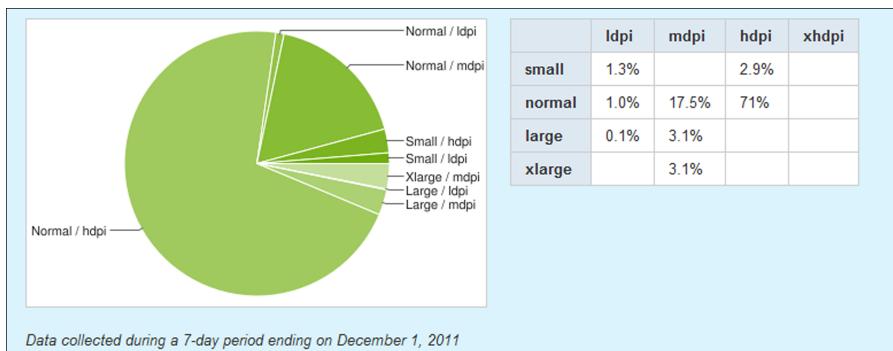


Figure 15.3 Android device statistics regarding screen sizes and densities (source: <http://goo.gl/22LxY>).

Although fragmentation makes the Android app developer's life more complicated, it's still possible to develop for and support a variety of devices—even all devices—with a single application. When it comes to maximizing compatibility, you'll always want to use the following strategies:

- Whenever possible, choose the development option that is supported by the widest variety of devices. In many cases, you can detect device differences at runtime and provide different code paths to support different configurations. Just make sure you inform your quality assurance team of this sort of application logic so it can be understood and thoroughly tested.
- Whenever a development decision limits the compatibility of your application (for example, using an API that was introduced in a later API level or introducing a hardware requirement such as camera support), assess the risk and document this limitation. Determine whether you are going to provide an alternative solution for devices that do not support this requirement.
- Consider screen size and resolution differences when designing application user interfaces. It is often possible to design very flexible layouts that look reasonable in both portrait and landscape modes, as well as different screen resolutions and sizes. However, if you don't consider this early, you will likely have to make changes (sometimes painful ones) later on to accommodate these differences.
- Test on a wide range of devices early in the development process to avoid unpleasant surprises late in the game. Make sure the devices have different hardware and software, including different versions of the Android platform, different screen sizes, and different hardware capabilities.
- Whenever necessary, provide alternative resources to help smooth over differences between device characteristics (we talk extensively about alternative resources later in this chapter).
- If you do introduce software and hardware requirements to your application, make sure you register this information in the Android manifest file using the appropriate tags. These tags, used by the Android platform as well as third parties such as the Android Market, help ensure that your application is only installed on devices that are capable of meeting your application's requirements.

Now let's look at some of the strategies you can use to target different device configurations and languages.

## Designing User Interfaces for Compatibility

Before we show you the many ways in which you can provide custom application resources and code to support specific device configurations, it's important to remember that you can often avoid needing them in the first place. The trick is to design your initial default solution to be flexible enough to cover any variations. When it comes to user

interfaces, keep them simple and don't overcrowd them. Also, take advantage of the many powerful tools at your disposal:

- As a rule of thumb, design for medium- to large-size screens and medium resolutions. Over time, devices trend toward larger screens with higher resolutions.
- Use `Fragments` to keep your screen designs independent from your application `Activity` classes and provide for flexible workflows. Leverage the Android Support Package to provide newer support libraries to older platform versions.
- For `View` and `Layout` control width and height attributes, use `match_parent` (also called the deprecated `fill_parent`) and `wrap_content` so that controls scale for different screen sizes and orientation changes, instead of using fixed pixel sizes.
- For dimensions, use the flexible units, such as `dp` and `sp`, as opposed to fixed-unit types, such as `px`, `mm`, and `in`.
- Avoid using `AbsoluteLayout` and other pixel-perfect settings and attributes.
- Use flexible layout controls such as `RelativeLayout`, `LinearLayout`, `TableLayout`, and `FrameLayout` to design a screen that looks great in both portrait and landscape modes and on a variety of different screen sizes and resolutions. Try the working square principle for organizing screen content—we talk more about this in a moment.
- Encapsulate screen content in scalable container controls such as `ScrollView` and `ListView`. Generally, you should scale and grow screens in only one direction (vertically or horizontally), but not both.
- Don't provide exact position values for screen elements, sizes, and dimensions. Instead, use relative positions, weights, and gravity. Spending time upfront to get these right saves time later.
- Provide application graphics of reasonable quality and always keep the original (larger) sizes around in case you need different versions for different resolutions at a later time. There is always a tradeoff in terms of graphic quality versus file size. Find the sweet spot where the graphic scales reasonably well for changes in screen characteristics, without bulking up your application or taking too long to display. Whenever possible, use stretchable graphics, such as Nine-Patch, which allow a graphic to change size based on the area in which it is displayed.



### Tip

Looking for information about the device screen? Check out the `DisplayMetrics` utility class, which, when used in conjunction with the window manager, can determine all sorts of information about the display characteristics of the device at runtime:

```
DisplayMetrics currentMetrics = new DisplayMetrics();
WindowManager wm = getWindowManager();
wm.getDefaultDisplay().getMetrics(currentMetrics);
```

## Working with Fragments

Fragments were discussed in detail in Chapter 10, “Working with Fragments,” but they deserve another mention here, when it comes to designing compatible applications. All applications can benefit from the screen workflow flexibility provided by fragment-based designs. By decoupling screen functionality from specific `Activity` classes, you have the option of pairing up that functionality in different ways, depending on the screen size, orientation, and other hardware configuration options. As new types of Android devices hit the market, you’ll be well placed for supporting them if you do this work upfront—in short, future-proofing your user interfaces.

### Tip

There’s little excuse not to use fragments, even if you are supporting legacy Android versions as far back as Android 1.6 (99% of the market). Simply use the Android Support Package to include these features in your legacy code. It’s just a right-click away in Eclipse. With most non-fragment-based APIs deprecated, it’s clearly the path along which the platform designers are leading developers.



## Leveraging the Android Support Package

Fragments and several other new features of the Android SDK (such as loaders) are so important for future device compatibility that there are now Android Support Packages to bring these APIs to older device platform versions, as far back as Android 1.6. To use the Android Support Package with your application, take the following steps:

1. Use the Android SDK Manager to download the Android Support Package.
2. Find your project in the Package Explorer or Project Explorer.
3. Right-click the project and choose `Android Tools, Add Compatibility Library...`. The most updated library will be downloaded and your project settings will be modified to use the newest library.
4. Begin using the APIs available as part of the Android Support Package. For example, to create a class extending `FragmentActivity`, you’d need to import `android.support.v4.app.FragmentActivity`.

## Supporting Specific Screen Types

Although you generally want to try to develop your applications to be screen independent (support all types of screens, small and large, high density and low), you can specify the types of screens your application can support explicitly when necessary in the Android manifest file. Here are some of the basics for supporting different screen types within your application:

- Explicitly state which screen sizes your application supports using the <supports-screens> Android manifest file tag. For more information on this Android manifest tag, see: <http://d.android.com/guide/topics/manifest/supports-screens-element.html>.
- Design flexible layouts that work with different size screens.
- Provide the most flexible default resources you can and add appropriate alternative layout and drawable resources for different screen sizes, densities, aspect ratios, and orientations as needed.
- Test, test, test! Make sure you review how your application behaves on devices with different screen sizes, densities, aspect ratios, and orientations regularly as part of your quality assurance testing cycle.



### Tip

For a very detailed discussion of how to support different types of screens, from the smallest smartphones to the largest tablets and televisions, see the Android Developer website: [http://d.android.com/guide/practices/screens\\_support.html](http://d.android.com/guide/practices/screens_support.html).

It's also helpful to understand how legacy applications are automatically scaled for larger and newer devices using what is called *screen compatibility mode*. Depending on the version of the Android SDK that your application originally targeted, the behavior on newer platform versions may be subtly different. This mode is on by default but can be disabled by your application. Learn more about screen compatibility mode at the Android Developer website: <http://d.android.com/guide/practices/screen-compat-mode.html>.

## Working with Nine-Patch Stretchable Graphics

Phone screens come in various dimensions. It can be handy to use stretchable graphics to allow a single graphic that can scale appropriately for different screen sizes and orientations or different lengths of text. This can save you a lot of time in creating graphics for many different screen sizes. Android supports Nine-Patch Stretchable Graphics for this purpose. Nine-Patch Stretchable Graphics are simply PNG graphics that have patches, or areas of the image, defined to scale appropriately, instead of scaling the entire image as one unit. We discussed how to create stretchable graphics in Chapter 4, "Mastering the Android Development Tools."

## Using the Working Square Principle

Another way to design for different screen orientations is to try to keep a "working square" area where most of your application's user activity (meaning, where they look and click on the screen) takes place. This area remains unchanged (or changes little

beyond just rotating) when the screen orientation changes. Only functionality displayed outside of the “working square” changes substantially when screen orientation changes (see Figure 15.4).

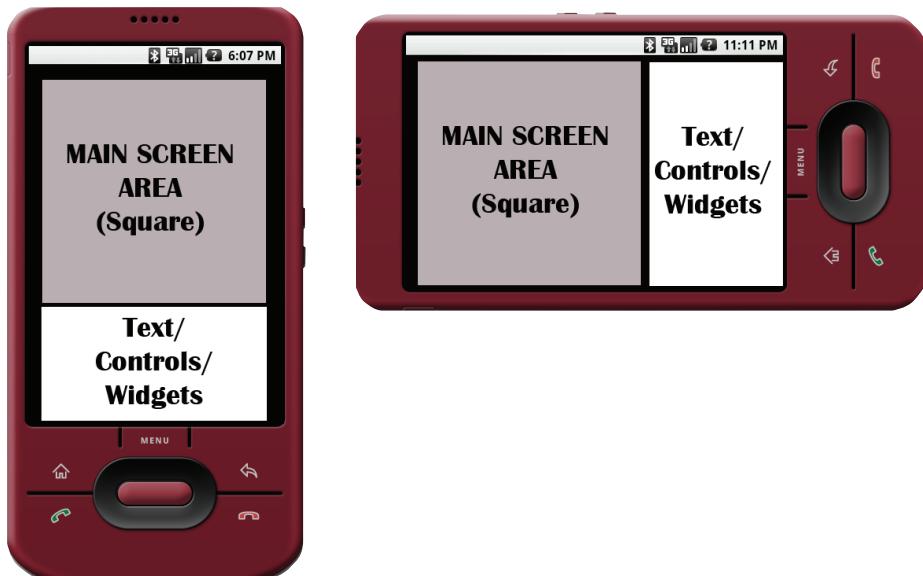


Figure 15.4 The “working square” principle.

One clever example of a “working square,” which turns the idea on its head, is the Camera application on the HTC Evo 4G. In Portrait mode, the camera controls are on the bottom of the viewfinder (see Figure 15.5, left); when rotated clockwise into Landscape mode, the camera controls stay in the same place but now they are to the left of the viewfinder (see Figure 15.5, right). The viewfinder area would be considered the working square—the area that remains uncluttered. The controls and sliding drawer with settings are kept outside that area, so the user can compose his or her photos and videos.

When you’re using the application, visually the rotation looks as if it has had little effect. The controls moved from being below the viewfinder to being to the left of the viewfinder. It just so happens, though, that they remain in the same location on the screen. This is part of the elegance of the “working square” principle.

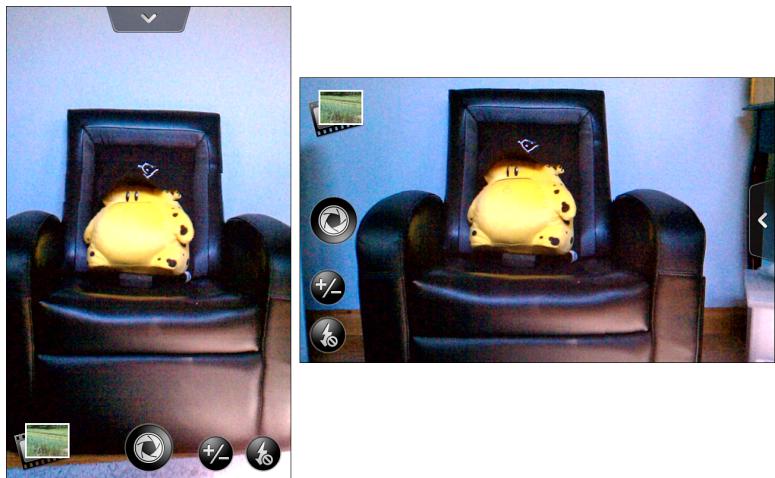


Figure 15.5 Evo 4G Camera application using a form of the “working square” principle.

## Providing Alternative Application Resources

Few application user interfaces look perfect on every device. Most require some tweaking and some special case handling. The Android platform allows you to organize your project resources so that you can tailor your applications to specific device criteria. It can be useful to think of the resources stored at the top of the resource hierarchy naming scheme as *default resources* and the specialized versions of those resources as *alternative resources*.

Here are some reasons you might want to include alternative resources within your application:

- To support different user languages and locales
- To support different device screen sizes, densities, dimensions, orientations, and aspect ratios
- To support different device docking modes
- To support different device input methods
- To provide different resources depending on the device’s Android platform version

## Understanding How Resources Are Resolved

Here’s how it works. Each time a resource is requested within an Android application, the Android operating system attempts to match the best possible resource for the job.

In many cases, applications provide only one set of resources. Developers can include alternative versions of those same resources as part of their application packages. The Android operating system always attempts to load the most specific resources available—the developer does not have to worry about determining which resources to load because the operating system handles this task.

There are four important rules to remember when creating alternative resources:

1. The Android platform always loads the most-specific, most-appropriate resource available. If an alternative resource does not exist, the default resource is used. Therefore, it's important to know your target devices, design for the defaults, and add alternative resources judiciously in order to keep your projects manageable.
2. Alternative resources must always be named exactly the same as the default resources and stored in the appropriately named directory, as dictated by a special alternative resource qualifier. If a string is called `strHelpText` in the `/res/values/strings.xml` file, then it must be named the same in the `/res/values-fr/strings.xml` (French) and `/res/values-zh/strings.xml` (Chinese) string files. The same goes for all other types of resources, such as graphics or layout files.
3. Good application design dictates that alternative resources should almost always have a default counterpart so that regardless of the device configuration, some version of the resource will always load. The only time you can get away without a default resource is when you provide every kind of alternative resource. One of the first steps the system takes when finding a best matching resource is to eliminate resources that are contradictory to the current configuration. For example, in portrait mode, the system would not even attempt to use a landscape resource, even if that is the only resource available. Keep in mind that new alternative resource qualifiers are added over time, so although you might think your application provides complete coverage of all alternatives now, it might not in the future.
4. Don't go overboard creating alternative resources because they add to the size of your application package and can have performance implications. Instead, try to design your default resources to be flexible and scalable. For example, a good layout design can often support both landscape and portrait modes seamlessly—if you use appropriate layouts, user interface controls, and scalable graphic resources.

## Organizing Alternative Resources with Qualifiers

Alternative resources can be created for many different criteria, including, but not limited to, screen characteristics, device input methods, and language or regional differences. These alternative resources are organized hierarchically within the `/res` resource project directory. You use directory qualifiers (in the form of directory name suffixes) to specify a resource as an alternative resource to load in specific situations.

A simple example might help to drive this concept home. The most common example of when alternative resources are used has to do with the default application icon resources created as part of a new Android project in Eclipse. An application could simply provide a single application icon graphic resource, stored in the `/res/drawable` directory. However, different Android devices have different screen resolutions. Therefore, alternative resources are used instead: `/res/drawable-hdpi/icon.png` is an application icon suitable for high-density screens, `/res/drawable-ldpi/icon.png` is the application icon suitable for low-density screens, and so on. Note that in each case, the alternative resource is named the same. This is important. Alternative resources must use the same names as the default resources. This is how the Android system can match the appropriate resource to load—by its name.

Here are some additional important facts about alternative resources:

- Alternative resource directory qualifiers are always applied to the default resource directory name. For example, `/res/drawable-qualifier`, `/res/values-qualifier`, `/res/layout-qualifier`.
- Alternative resource directory qualifiers (and resource filenames) must always be lowercase, with one exception: region qualifiers.
- Only one directory qualifier of a given type may be included in a resource directory name. Sometimes this has unfortunate consequences—you might be forced to include the same resource in multiple directories. For example, you cannot create an alternative resource directory called `/res/drawable-1dpi-mdpi` to share the same icon graphic. Instead, you must create two directories: `/res/drawable-1dpi` and `/res/drawable-mdpi`. Frankly, when you want different qualifiers to share resources instead of providing two copies of the same resource, you’re often better off making those your default resources and then providing alternative resources for those that do not match `1dpi` and `mdpi`—that is, `hdpi`. As we said, it’s up to you how you go about organizing your resources; these are just our suggestions for keeping things under control.
- Alternative resource directory qualifiers can be combined or chained, with each qualifier being separated by a dash. This enables developers to create very specific directory names and therefore very specialized alternative resources. These qualifiers must be applied in a very specific order, and the Android operating system always attempts to load the most specific resource (that is, the resource with the longest matching path). For example, you can create an alternative resource directory for French language (qualifier `fr`), Canadian region (qualifier `rCA`—this is a region qualifier and is therefore capitalized) string resources (stored in the `values` directory) as follows: `/res/values-fr-rCA/strings.xml`.

- You only need to create alternative resources for the specific resources you require—not every resource in a given file. If you only need to translate half the strings in the default `strings.xml` file, then only provide alternative strings for those specific string resources. In other words, the default `strings.xml` resource file might contain a superset of string resources with the alternative string resource files containing a subset—only the strings requiring translation. Common examples of strings that do not get localized are company or brand names.
- No custom directory names or qualifiers are allowed. You may only use the qualifiers defined as part of the Android SDK. These qualifiers are listed in Table 15.1.
- Always try to include default resources—that is, those resources saved in directories without any qualifiers. These are the resources that the Android operating system will fall back on when no specific alternative resource matches the criteria. If you don't, the system falls back on the closest matching resource based upon the directory qualifiers—one that might not make sense.

Now that you understand how alternative resources work, let's look at some of the directory qualifiers you can use to store alternative resources for different purposes. Qualifiers are tacked on to the existing resource directory name in a strict order, shown in descending order in Table 15.1.

Good examples of alternative resource directories with qualifiers are

- `/res/values-en-rUS-port-finger`
- `/res/drawables-en-rUS-land-mdpi`
- `/res/values-en-qwerty`

Bad examples of alternative resource directories with qualifiers are

- `/res/values-en-rUS-rGB`
- `/res/values-en-rUS-port-FINGER-wheel`
- `/res/values-en-rUS-port-finger-custom`
- `/res/drawables-rUS-en`

The first bad example does not work because you can have only one qualifier of a given type, and this one violates that rule by including both `rUS` and `rGB`. The second bad example violates the rule that qualifiers (with the exception of the Region) are always lowercase. The third bad example includes a custom attribute defined by the developer, but these are not currently supported. The last bad example violates the order in which the qualifiers must be placed: Language first, then Region, and so on.

Table 15.1 Important Alternative Resource Qualifiers

Directory Qualifier	Example Values	Description
Mobile country code and mobile network code	mcc310 (United States) mcc310-mnc004 (United States, Verizon) mcc208-mnc00 (France, Orange)	The mobile country code (MCC), optionally followed by a dash and a mobile network code (MNC) from the SIM card in the device.
Language and region code	en (English)  ja (Japanese) de (German) en-rUS (American English) en-rGB (British English)	The language code (ISO 639-1 two-letter language code), optionally followed by a dash and the region code (a lowercase “r” followed by the region code as defined by ISO 3166-1-alpha-2)
Screen pixel dimensions. Several qualifiers for specific screen dimensions, including smallest width, available width, and available height	sw<N>dp (Smallest width) w<N>dp (Available width) h<N>dp (Available height)  Examples: sw320dp sw480dp sw600dp sw720dp h320dp h540dp h800dp w480dp w720dp w1080dp	DP-specific screen requirements for your application. swXXXdp: Indicates the smallest width that this resource qualifier supports. wYYYdp: Indicates the minimum width. hZZZdp: Indicates the minimum height. The numeric value can be any width the developer desires, in dp units. Added in API Level 13.

Directory Qualifier	Example Values	Description
Screen size	small normal large xlarge (Added in API Level 9)	Generalized screen size. A small screen is generally a low-density QVGA or higher density VGA screen. A normal screen is generally a medium-density HVGA screen or similar. A large screen has at least a medium-density VGA screen or other screen with more pixels than an HVGA display. An xlarge screen has at least a medium-density HVGA screen and is generally tablet sized or larger. Added in API Level 4.
Screen aspect ratio	long notlong	Whether or not the device is a wide-screen device. WQVGA, WVGA, FWVGA screens are long screens. QVGA, HVGA, and VGA screens are notlong screens. Added in API Level 4.
Screen orientation	port land	When a device is in portrait mode, the port resources will be loaded. When the device is in landscape mode, the land resources will be loaded.
Dock mode	car desk	Load specific resources when the device is in a car or desk dock. Added in API Level 8.
Night mode	night notnight	Load specific resources when the device is in night mode or not. Added in API Level 8.

Table 15.1 Continued

Directory Qualifier	Example Values	Description
Screen pixel density	ldpi mdpi hdpi xhdpi (Added in API Level 8) tvdpi (Added in API Level 13) nodpi	<p>Low-density screen resources (approx. 120dpi) should use the <code>ldpi</code> option.</p> <p>Medium-density screen resources (approx. 160dpi) should use the <code>mdpi</code> option.</p> <p>High-density screen resources (approx. 240dpi) should use the <code>hdpi</code> option.</p> <p>Extra high-density screen resources (approx. 320dpi) should use the <code>xhdpi</code> option.</p> <p>Television screen resources (approx. 213dpi, between <code>mdpi</code> and <code>hdpi</code>) should use the <code>tvdpi</code> option.</p> <p>Use the <code>nodpi</code> option to specify resources that you do not want to be scaled to match the screen density of the device.</p> <p>Added in API Level 4.</p>
Touch screen type	notouch stylus finger	<p>Resources for devices without touch screens should use the <code>notouch</code> option.</p> <p>Resources for devices with a stylus-style (resistive) touch screen should use the <code>stylus</code> option.</p> <p>Resources for devices with finger-style (capacitive) touch screens should use the <code>finger</code> option.</p>
Keyboard type and availability	keysexposed keyshidden keyssoft	<p>Use the <code>keysexposed</code> option for resources when a keyboard is available (hardware or soft keyboard).</p> <p>Use the <code>keyshidden</code> option for resources when no hardware or software keyboard is available.</p> <p>Use the <code>keyssoft</code> option for resources when the soft keyboard is available.</p>

Directory Qualifier	Example Values	Description
Text input method	nokeys qwerty 12key	Use the <code>nokeys</code> option for resources when the device has no hardware keys for text input. Use the <code>qwerty</code> option for resources when the device has a QWERTY hardware keyboard for text input. Use the <code>12key</code> option for resources when the device has a 12-key numeric keypad for text input.
Navigation key availability	navexposed navhidden	Use <code>navexposed</code> for resources when the navigational hardware buttons are available to the user. Use <code>navhidden</code> for resources when the navigational hardware buttons are not available to the user (such as when the phone case is slid shut).
Navigation method	nonav dpad trackball wheel	Use <code>nonav</code> if the device has no navigation buttons other than a touch screen. Use <code>dpad</code> for resources where the primary nav method is a directional pad. Use <code>trackball</code> for resources where the primary nav method is a trackball. Use <code>wheel</code> for resources where the primary nav method is a directional wheel.
Android platform	v3 (Android 1.5) v4 (Android 1.6) v7 (Android 2.1.X) v8 (Android 2.2.X) v9 (Android 2.3–2.3.2) v10 (Android 2.3.3–2.3.4) v12 (Android 3.1.X) v13 (Android 3.2.X) v14 (Android 4.0.X)	Load resources based on the Android platform version, as specified by the API level. This qualifier will load resources for the specified API level or higher. Note: There are some known issues for this qualifier. See the Android documentation for details.

## Providing Resources for Different Orientations

Let's look at a very simple application that uses alternative resources to customize screen content for different orientations. The SimpleAltResources application (see the book's sample code for a complete implementation) has no real code to speak of—check the `Activity` class if you don't believe us. Instead, all interesting functionality depends on the resource folder qualifiers. These resources are

- The default resources for this application include the application icon and a picture graphic stored in the `/res/drawable` directory, the layout file stored in the `/res/layout` directory, and the color and string resources stored in the `/res/values` directory. These resources are loaded whenever a more specific resource is not available to load. They are the fallbacks.
- There is a portrait-mode alternative picture graphic stored in the `/res/drawable-port` directory. There are also portrait-mode-specific string and color resources stored in the `/res/values-port` directory. If the device is in portrait orientation, these resources—the portrait picture graphic, the strings, and colors—are loaded and used by the default layout.
- There is a landscape-mode alternative picture graphic stored in the `/res/drawable-land` directory. There are landscape-mode-specific string and color (basically reversed background and foreground colors) resources stored in the `/res/values-land` directory as well. If the device is in landscape orientation, these resources—the landscape picture graphic, the strings, and the colors—are loaded and used by the default layout.

Figure 15.6 illustrates how the application loads different resources based on the orientation of the device at runtime. This figure shows the project layout, in terms of resources, as well as what the screen looks like in different device orientations.

## Using Alternative Resources Programmatically

There is currently no easy way to request resources of a specific configuration programmatically. For example, the developer cannot programmatically request the French or English version of the string resource. Instead, the Android system determines the resource at runtime, and developers refer only to the general resource variable name.

## Organizing Application Resources Efficiently

It's easy to go too far with alternative resources. You could provide custom graphics for every different permutation of device screen, language, or input method. However, each time you include an application resource in your project, the size of your application package grows.



Figure 15.6 Using alternative resources for portrait and landscape orientations.

There are also performance issues with swapping out resources too frequently—generally when runtime configuration transitions occur. Each time a runtime event such as an orientation or keyboard state change occurs, the Android operating system restarts the underlying `Activity` and reloads the resources. If your application is loading a lot of resources and content, these changes come at a cost to application performance and responsiveness.

Choose your resource organization scheme carefully. Generally, you should put the most commonly used resources as your defaults and then carefully overlay alternative resources only when necessary. For example, if you are writing an application that routinely shows videos or displays a game screen, you might want to make landscape mode resources your defaults and provide alternative portrait mode resources because they are not as likely to be used.

### Retaining Data Across Configuration Changes

An Activity can keep data around through these transitions by using the `onRetainNonConfigurationInstance()` method to save data and the `getLastNonConfigurationInstance()` method to restore this data after the transition. This functionality can be especially helpful when your Activity has a lot of setup or preloading to do.

### Handling Configuration Changes

In cases where your Activity does not need to reload alternative resources on a specific transition, you might want to consider having the Activity class handle the transition to avoid having your Activity restart. The camera application mentioned earlier could use this technique to handle orientation changes without having to reinitialize the camera hardware internals, redisplay the viewfinder window, or redisplay the camera controls (the Button controls simply rotate in place to the new orientation—very slick).

For an Activity class to handle its own configuration changes, your application must

- Update the `<activity>` tag in the Android manifest file for that specific Activity class to include the `android:configChanges` attribute. This attribute must specify the types of changes the Activity class handles itself.
- Implement the `onConfigurationChanged()` method of the Activity class to handle the specific changes (by type).

## Targeting Tablets, TVs, and Other New Devices

The past year or so has seen tremendous growth in the types of devices supported by the Android platform. Whether we're talking tablets, TVs, or toasters, there's something for everyone. These new devices make for an exciting time for application developers. New devices mean new groups and demographics of users using the platform. These new types of Android devices pose some unique challenges for Android developer.

### Targeting Tablet Devices

Tablets are this year's hottest new type of Android device. They come in a variety of sizes and default orientations, from many different manufacturers and carriers. Luckily, from a developer's perspective, tablets can be considered just another Android device, provided that you haven't made any unfortunate development assumptions.

Android tablets run the same platform versions that traditional smartphones do—there is nothing special. Most tablets these days are running Android 2.0 and higher. The latest crop of popular tablets run primarily Gingerbread and Honeycomb, which have a lot of new features for handling different screen sizes, resolutions, and navigational mechanisms. Ice Cream Sandwich combines the smartphone-centric SDK features of Froyo and Gingerbread with the smart-device features of Honeycomb, so that we have one SDK that really suits all Android devices for the future.

Here are some tips for designing, developing, and publishing Android applications for the tablet devices:

- **Design flexible user interfaces:** Regardless of what devices your applications are targeting, use flexible layout designs. Use `RelativeLayout` to organize your user interfaces. Use relative dimension values such as `dp` instead of specific ones such as `px`. Use stretchable graphics such as Nine-Patch.
- **Take advantage of fragments:** Fragments make for much more flexible user interface navigation by decoupling screen functionality from specific activities.
- **Leverage alternative resources:** Provide alternative resources for various device screen sizes and densities.
- **Screen orientation:** Tablets often default to landscape mode, but this is not always the case. Some tablets, especially smaller ones, use portrait defaults.
- **Input mode limitations:** Tablets often rely solely on touch screen input. Some configurations also have a few other physical buttons, but this is unusual because Honeycomb, the first platform revision to truly support tablets, moved the typical hardware buttons (Back, Home, Search, and Menu) to the touchscreen.
- **UI navigational differences:** Users hold and tap on tablets in a different fashion than they do smartphones. In portrait and landscape modes, tablet screens are substantially wider than their smartphones equivalents. Applications such as games that rely on the user cradling the device in their hands like a traditional game controller may struggle with all the extra room on a tablet. The user's thumbs might easily reach or access the two halves of a smartphone screen but not be able to do the same on a tablet.
- **Feature support:** Certain hardware and software features are not usually available on tablets. For example, telephony is not always available. This has implications on unique devices identifiers; many used to rely on the telephony identifier that may not be present. Point is, hardware differences can also lead to other, less obvious impacts.

## Targeting Google TV Devices

Google TV is a relatively new type of device that Android developers can now target. Users can browse the Android Market for compatible applications and download them much like they would to other Android devices.

In order to develop Google TV applications, developers use the Android SDK as well as a Google TV add-on, which can be downloaded using the Android SDK Manager.

There are some subtle differences between development for Google TV devices and targeting smartphones and tablets. Let's look at some development tips for targeting Google TV devices:

- **Screen density and resolution:** Google TV devices currently run at two resolutions. The first is 720p (aka “HD” and tvdpi), or 1280 × 720 pixels. The second is 1080p (aka “Full HD” and xhdpi), or 1920 × 1080 pixels. These correspond to large screen size.
- **Screen orientation:** Google TV devices only need landscape-oriented layouts.
- **Not pixel perfect:** One caveat to Google TV development is to not rely on the exact number of pixels on the screen. Televisions don’t always expose every single pixel. Therefore, your screen designs should be flexible enough to accommodate small adjustments when you’ve had a chance to test your applications on real Google TV devices. Use of `RelativeLayout` is highly recommended. See <http://goo.gl/fgNFF> for more information on this issue.
- **Input mode limitations:** Unlike tablets or smartphones, Google TV devices are not within arm’s reach and do not have touch screens. This means no gestures, no multitouch, and so on. The Google TV interface uses a directional pad (or d-pad)—that is, arrow keys for up, down, left, and right along with a Select button and Media keys (Play, Pause, and so on). Some configurations also have a mouse or keyboard.
- **UI navigational differences:** The input type limitations with Google TV devices may mean you need to make some changes to your application’s screen navigation. Users can’t easily skip over focusable items on the screen. For instance, if your UI has a row of items, with the two most common on the far left and far right for convenient access with thumb clicks, these items may be inconveniently separated for the average Google TV user.
- **Android manifest file settings:** A number of Android manifest file settings should be configured appropriately for the Google TV. Review the Google TV documentation for details:  
[http://code.google.com/tv/android/docs/gtv\\_androidmanifest.html](http://code.google.com/tv/android/docs/gtv_androidmanifest.html)
- **Android market filters:** The Android Market uses Android manifest file settings to filter applications and provide them to the appropriate devices. Certain features, such as those defined using the `<uses-feature>` tag, may exclude your application from Google TV devices. One example of this is when applications require features such as touch screen, camera, and telephony. For a complete list of features supported and unsupported by the Google TV, see the following page:  
[http://code.google.com/tv/android/docs/gtv\\_android\\_features.html](http://code.google.com/tv/android/docs/gtv_android_features.html).
- **Feature support:** Certain hardware and software features (sensors, cameras, telephony, and so on) are not available on Google TV devices.
- **No native development kit:** There is currently no NDK support for Google TV.

- **Supported media formats:** There are subtle differences between the media formats supported by the Android platform (<http://d.android.com/guide/appendix/media-formats.html>) and those supported on the Google TV platform ([http://code.google.com/tv/android/docs/gtv\\_media\\_formats.html](http://code.google.com/tv/android/docs/gtv_media_formats.html)).

 Tip

For more information on developing for Google TV devices, see the Google TV Android Developers Guide at <http://code.google.com/tv/android/>.

## Summary

Compatibility is a vast topic, and we've given you a lot to think about. During design and implementation, always consider if your choices are going to introduce roadblocks to device compatibility. Quality assurance personnel should always vary the devices used for testing as much as possible—certainly don't rely solely on emulator configurations for testing coverage. Use best practices that encourage compatibility and do your best to keep compatibility-related resources and code streamlined and manageable.

If you take only two concepts away from this chapter, they should be that alternative resources and fragments can be used to great effect. They enable a flexibility that can go a long way toward achieving compatibility, whether it's for screen differences or internationalization. Additionally, certain Android manifest file tags can help ensure that your applications are installed only on devices that meet certain prerequisites, or requirements, such as a certain version of OpenGL ES or the availability of camera hardware.

## References and More Information

Android SDK Reference regarding the application Dialog class:

<http://d.android.com/reference/android/app/Dialog.html>

Android SDK Reference regarding the Android Support Package:

<http://d.android.com/sdk/compatibility-library.html>

Android Dev Guide: "Screen Compatibility Mode":

<http://d.android.com/guide/practices/screen-compat-mode.html>

Android Dev Guide: "Providing Alternative Resources":

<http://d.android.com/guide/topics/resources/providing-resources.html#AlternativeResources>

Android Dev Guide: "How Android Finds the Best-Matching Resource":

<http://d.android.com/guide/topics/resources/providing-resources.html#BestMatch>

Android Dev Guide: "Handling Runtime Changes":

<http://d.android.com/guide/topics/resources/runtime-changes.html>

Android Best Practices: “Compatibility”:

<http://d.android.com/guide/practices/compatibility.html>

Android Best Practices: “Supporting Multiple Screens”:

[http://developer.android.com/guide/practices/screens\\_support.html](http://developer.android.com/guide/practices/screens_support.html)

ISO 639-1 Languages:

[http://www.loc.gov/standards/iso639-2/php/code\\_list.php](http://www.loc.gov/standards/iso639-2/php/code_list.php)

ISO 3166-1-alpha-2 Regions:

[http://www.iso.org/iso/country\\_codes/iso\\_3166\\_code\\_lists](http://www.iso.org/iso/country_codes/iso_3166_code_lists)



# Publishing and Distributing Android Applications

- 16** The Android Software Development Process
- 17** Designing and Developing Bulletproof Android Applications
- 18** Testing Android Applications
- 19** Publishing Your Android Application

*This page intentionally left blank*

# The Android Software Development Process

The mobile development process is similar to the traditional desktop software process, with a couple of distinct differences. Understanding how these differences affect your mobile development team is critical to running a successful project. This insight into the mobile development process is invaluable to those new to mobile development and veteran developers alike, to those in management and planning, as well as to the developers and testers in the trenches. In this chapter, you learn about the peculiarities of mobile development as they pertain to each stage of the software development process.

## An Overview of the Mobile Development Process

Mobile development teams are often small in size and project schedules are short in length. The entire project lifecycle is often condensed, and whether you're a team of one or one hundred, understanding the mobile development considerations for each part of the development process can save you a lot of wasted time and effort. Some hurdles a mobile development team must overcome include

- Choosing an appropriate software methodology for your mobile project
- Understanding how target devices dictate the functionality of your application
- Performing thorough, accurate, and ongoing feasibility analyses
- Mitigating the risks associated with preproduction devices
- Keeping track of device functionality through configuration management
- Designing a responsive, stable application on a memory-restrictive system
- Designing user interfaces for a variety of devices with different user experiences
- Testing the application thoroughly on the target devices
- Incorporating third-party requirements that affect where you can sell your application
- Deploying and maintaining a mobile application

## Choosing a Software Methodology

Developers can easily adapt most modern software methodologies to mobile development. Whether your team opts for traditional Rapid Application Development (RAD) principles or more modern variants of agile software development, such as Scrum, mobile applications have some unique requirements.

### Understanding the Dangers of Waterfall Approaches

The short development cycle might tempt some to use a waterfall approach, but developers should beware of the inflexibility that comes with this choice. It is generally a bad idea to design and develop an entire mobile application without taking into account the many changes that tend to occur during the development cycle (see Figure 16.1). Changes to target devices (especially preproduction models, but sometimes shipping devices can be substantially changed in software), ongoing feasibility, and performance concerns and the need for quality assurance to test early and often on the target devices (not just the emulator) make it difficult for strict waterfall approaches to succeed with mobile projects.

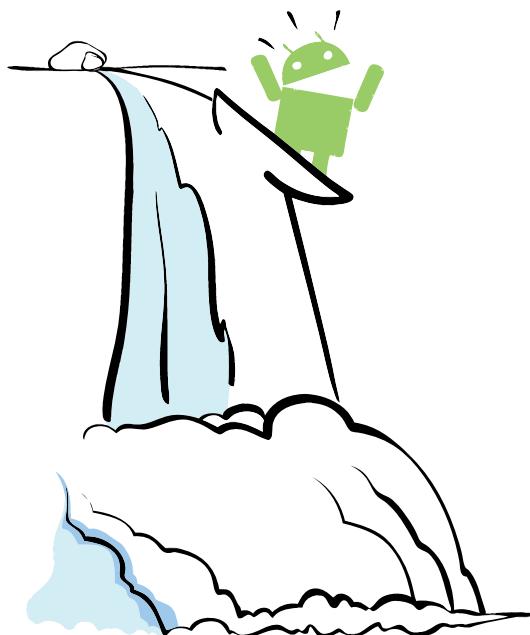


Figure 16.1 The dangers of waterfall development.  
(Graphic courtesy of Amy Tam Badger.)

## Understanding the Value of Iteration

Because of the speed at which mobile projects tend to progress, iterative methods have been the most successful strategies adapted to mobile development. Rapid prototyping enables developers and quality assurance personnel ample opportunity to evaluate the feasibility and performance of the mobile application on the target devices and adapt as needed to the change that inevitably occurs over the course of the project.

## Gathering Application Requirements

Despite the relative simplicity of a mobile application's feature set compared to a traditional desktop application, requirements analyses for a mobile application can be more complex. The mobile user interface must be elegant and the application must be fault tolerant, not to mention responsive in a resource-constrained environment. You must often tailor requirements to work across a number of devices—devices that might have vastly different user interfaces and input methods. Having great variation in target platforms can make development assumptions tricky. It's not unlike the differences web developers might need to accommodate when developing for different web browsers (and versions of web browsers).

## Determining Project Requirements

When multiple devices are involved (which is almost always the case with Android), we have found several approaches to be helpful for determining project requirements. Each approach has its benefits and its drawbacks. These approaches are

- The lowest common denominator method
- The customization method

### Using the Lowest Common Denominator Method

With the lowest common denominator method, you design the application to run *sufficiently* well across a number of devices. In this case, the primary target for which you develop is the device configuration with the fewest features—basically, the most inferior device. Only requirements that can be met by all devices are included in the specification in order to reach the broadest range of devices—requirements such as input methods, screen resolution, and the platform version. With this method, you'll often put a stake in the ground for a specific Android API level and then tailor your application further using Android manifest file settings and market filters.



### Note

The lowest common denominator method is roughly equivalent to developing a desktop application with the following minimum system requirements: (1) Windows 2000 and (2) 128 megabytes of RAM, on the assumption that the application will be forward compatible with the latest version of Windows (and every other version in between). It's not ideal, but in some cases, the trade-offs are acceptable.

Some light customization, such as resources and the final compiled binary (and the version information) is usually feasible with the lowest common denominator method. The main benefit of this method is that there is only one major source code tree to work with; bugs are fixed in one place and apply for all devices. You can also easily add other devices without changing much code, provided they, too, meet the minimum hardware requirements. The drawbacks include the fact that the resulting generalized application does not maximize any device-specific features, nor can it take advantage of new platform features. Also, if a device-specific problem arises or you misjudge the lowest common denominator and later find that an individual device lacks the minimum requirements, the team might be forced to implement a workaround (hack) or branch the code at a later date, losing the early benefits of this method but keeping all the drawbacks.



### Tip

The Android SDK makes it easy for developers to target multiple platform versions within a single application package. Developers should take care to identify target platforms early in the design phase. That said, over-the-air firmware updates to users are a fairly regular occurrence, so the platform version on a given device is likely to change over time. Always design your applications with forward compatibility in mind and make contingency plans for distributing application upgrades to existing applications as necessary.

## Using the Customization Method

Using the customization method, the application is tailored for specific devices or a class of devices (such as all devices capable of OpenGL ES 2.0, for example). This method works well for specialized applications with a small number of target devices but does not scale easily from a build or product management perspective.

Generally, developers will come up with a core application framework (classes or packages) shared across all versions of the application. All versions of a client/server application would likely share the same server and interact with it in the same way, but the client implementation is tailored to take advantage of specific device features, when they are available. The primary benefit of this technique is that users receive an application that leverages all the features their device (or API level) has to offer. Some drawbacks include source code fragmentation (many branches of the same code), increased testing requirements, and the fact that it can be more difficult to add new devices in the future. Only recently has the Android Market allowed developers to tie multiple package

files together under a single product name; prior to this change, developers had to create a different product name for each separate package: For example, My Game for Tablets and My Game for Smartphones and My Game for Google TV were managed completely separately, regardless of how much code they shared. This allows developers to create optimal packages that don't contain a lot of resources not needed by every device. For example, a "small screen package" wouldn't need resources for tablets and televisions.

### Taking Advantage of the Best of Both Methods

In truth, mobile development teams usually use a hybrid approach incorporating some of the aspects from both methods. It's pretty common to see developers define classes of devices based on functionality. For example, a game application might group devices based on graphics performance, screen resolution, or input methods. A location-based service (LBS) application might group devices based on the available internal sensors. Other applications might develop one version for devices with built-in front-facing cameras and one version for those without. These groupings are arbitrary and set by the developer to keep the code and testing manageable. They will, in large part, be driven by the details of a particular application and any support requirements. In many cases, these features can be detected at runtime as well, but add enough of them together and the code paths can become overly complex when having two or more applications would actually be easier.

#### Tip

A single, unified version of an application is cheaper to support than multiple versions. However, a game might sell better with custom versions that leverage the distinct advantages and features of a specific class of devices. A vertical business application would likely benefit more from a unified approach that works the same, is easier to train users across multiple devices, and would thus have lower support costs for the business.

### Developing Use Cases for Mobile Applications

You should first write use cases in general terms for the application before adapting them to specific device classes, which impose their own limitations. For example, a high-level use case for an application might be "Enter Form Data," but the individual devices might use different input methods, such as hardware versus software keyboards, and so on.

#### Tip

Developing an application for multiple devices is much like developing an application for different operating systems and input devices (such as handling Mac keyboard shortcuts versus those on Windows)—you must account for subtle and not-so-subtle differences. These differences might be obvious, such as not having a keyboard for input, or not so obvious, such as device-specific bugs or different conventions for soft keys. See Chapter 15, "Designing Compatible Applications," for a discussion on device compatibility.

## Incorporating Third-Party Requirements

In addition to the requirements imposed by your internal requirements analyses, your team needs to incorporate any requirements imposed by others. Third-party requirements can come from any number of sources, including

- Android SDK License Agreement Requirements
- Google Maps API License Agreement Requirements (if applicable)
- Other Google Add-ons License Requirements (if applicable)
- Other Third-Party API Requirements (if applicable)
- Android Market Requirements (if applicable)
- Other Application Store Requirements (if applicable)
- Mobile Carrier/Operator Requirements (if applicable)
- Application Certification Requirements (if applicable)

Incorporating these requirements into your project plan early is essential not only for keeping your project on schedule but also so that these requirements are built in to the application from the ground up, as opposed to applied as an afterthought, which can be risky.

## Managing a Device Database

As your mobile development team builds applications for a growing number of devices, it becomes more and more important to keep track of the target device information for revenue estimation and maintenance purposes. Creating a device database is a great way to keep track of both marketing and device specification details for target devices. When we say “database,” we mean anything from a Microsoft Excel spreadsheet to a little SQL database. The point is that the information is shared across the team or company and kept up to date. It can also be helpful to break devices into classes, such as those that support OpenGL ES 2.0 or those without camera hardware.

The device database is best implemented early, when project requirements are just determined and target devices are determined. Figure 16.2 illustrates how you can track device information and how different members of the application development team can use it.



### Tip

We've been asked by readers for our take on using personal devices for testing purposes. Is it safe? Is it smart? The short answer is, you can use your personal device for testing safely in most cases. It's highly unlikely that you will "break" or "brick" your device to the point in any way that a factory reset won't fix. However, protecting your data is another problem entirely. For example, if your application acts on the Contacts database, your real contacts may be messed with, by bugs or other coding mistakes. Sometimes using personal devices is convenient, especially for small development teams without a big hardware budget. Make sure you understand the ramifications of doing so. You can read our full discussion of this issue over on our Android book blog: <http://goo.gl/Cm2ed>.

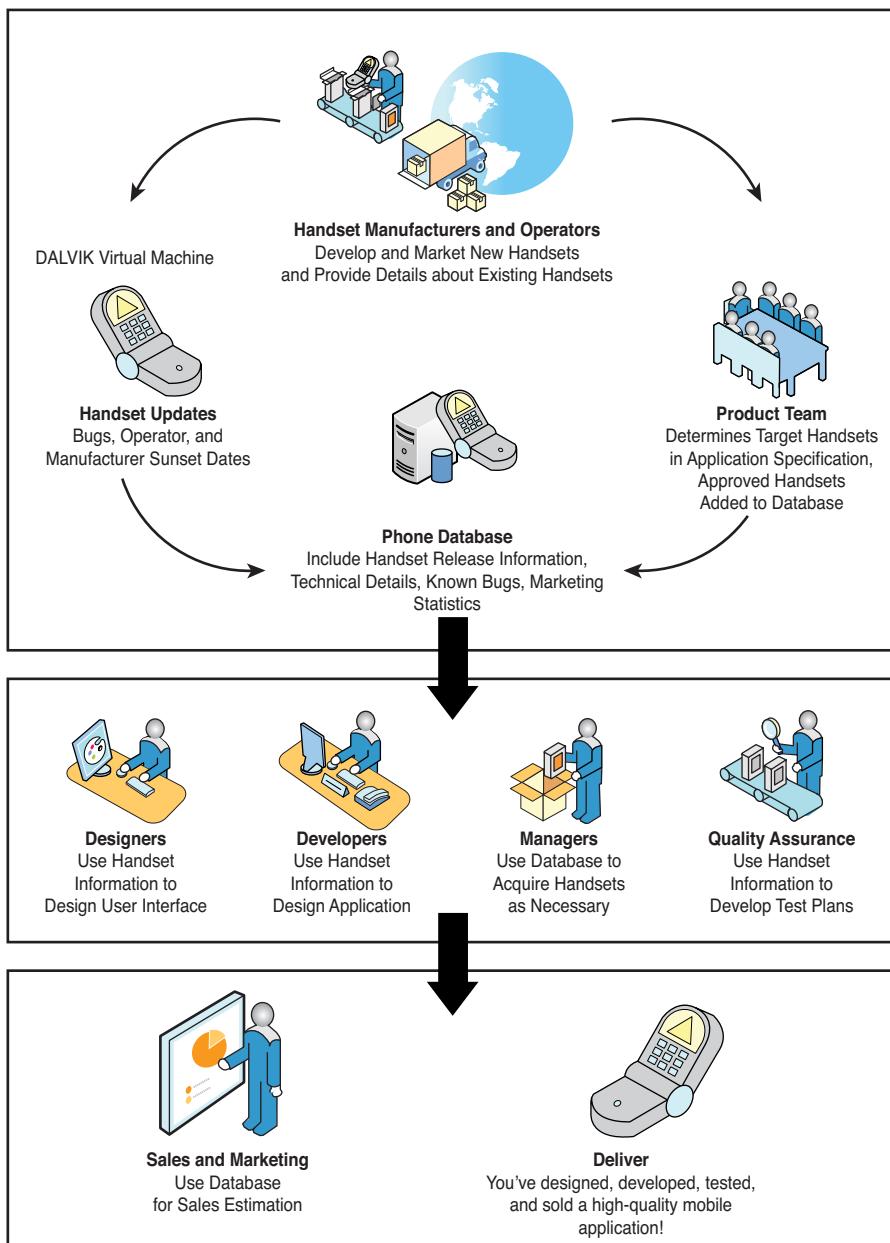


Figure 16.2 How a development team uses the device database.

## Determining Which Devices to Track

Some companies track only the devices they actively develop for, whereas others also track devices they might want to include in the future, or lower priority devices. You can include devices in the database during the Requirements phase of a project but also later as a change in project scope. You can also add devices as subsequent porting projects long after the initial application has been released.

## Storing Device Data

You should design the device database to contain any information about a given device that would be helpful for developing and selling applications. This might require that someone be tasked with keeping track of a continual stream of information from carrier and manufacturers. Still, this information can be useful for all mobile projects at a company. This data should include

- Important device technical specification details (screen resolution, hardware details, supported media formats, input methods, localization).
- Any known issues with devices (bugs and important limitations).
- Device carrier information (any firmware customizations, release and sunset dates, expected user statistics, such as if a device is highly anticipated and expected to sell a lot, is well received for vertical market applications, and so on).
- API level data and firmware upgrade information. (As information becomes available, changes might have no impact on the application or warrant an entirely separate device entry; keep in mind that different carriers roll out upgrades on different schedules, so keeping track of this information at a too-fine-grained level may not be feasible.)
- Actual testing device information (which devices have been purchased or loaned through manufacturer or carrier loaner programs, how many are available, and so on).

You can also cross-reference the device carrier information with sales figures from the carrier, application store, and internal metrics.

The actual testing device information is often best implemented as a library check-out system. Team members can reserve devices for testing and development purposes. When a loaner device needs to be returned to the manufacturer, it's easy to track. This also facilitates sharing devices across teams.

## Using Device Data

Remember that the database can be used for multiple mobile development projects. Device resources can be shared, and sales statistics can be compared to see on which devices your applications perform best. Different team members (or roles) can use the device database in different ways:

- Product designers use the database to develop the most appropriate application user interface for the target devices.
- Media artists use the database to generate application assets such as graphics, videos, and audio in supported media file formats and resolutions appropriate for the target devices.
- Project managers use the database to determine the devices that must be acquired for development and testing purposes on the project and development priorities.
- Software developers use the database to design and develop applications compatible with target device specifications.
- Quality assurance personnel use the database to design and develop the target device specifications for test plans and to test the application thoroughly.
- Marketing and sales professionals use the database to estimate sales figures for released applications. For example, it is important to be aware that application sales will drop as device availability drops.

The information in the database can also help determine the most promising target devices for future development and porting.

### **Using Third-Party Device Databases**

There are third-party databases for device information, including screen size and internal device details and carrier support details, but subscribing to such information can be costly for a small company. Many mobile developers instead choose to create a custom device database with only the devices they are interested in and the specific data they need for each device, which is often absent from open and free databases. WURFL (<http://wurfl.sourceforge.net>), for instance, is better for mobile web development rather than application development.

## **Assessing Project Risks**

In addition to the normal risks any software project must identify, mobile projects need to be aware of the outside influences that can affect their project schedule and whether the project requirements can be met. Some of the risk factors include identifying and acquiring target devices and continually reassessing application feasibility.

### **Identifying Target Devices**

Just as most sane software developers wouldn't write a desktop application without first deciding what operating systems (and their versions) the application would run on, mobile developers must consider the target devices their application will run on. Each device has different capabilities, a different user interface, and unique limitations.

Target devices are generally determined in one of two ways:

- There's a popular “killer” device you want to develop for.
- You want to develop an application for maximum coverage.

In the first instance, you have your initial target device (or class of devices) figured out. In the second instance, you want to look at the available (and soon-to-be-available) devices on the market and adjust your application specification to cover as many as is reasonably feasible.

#### Tip

On the Android platform, you normally do not target individual devices specifically, but device features or classes (for example, those running a specific platform version or having specific hardware configurations). You can limit the devices upon which your application will be installed using Android Manifest tags, which act as market filters.

### **Understanding How Manufacturers and Operators Fit In**

It's also important to note that we've seen popular product lines, such as the Droid line of Android devices, customized by a number of manufacturers. A carrier often ships its custom version of a device, including a different user experience or skin, as well as big bundles of custom applications (taking up a bunch of space on the device). The carrier might also disable specific device features (such as Bluetooth or Wi-Fi), which effectively makes it impossible for your application to run. You must take all these factors into account when considering your application requirements and capabilities. Your application's running requirements must match the features shared across all target devices and handle optional feature use appropriately in all cases.

### **Understanding How Devices Come and Go Over Time**

New devices are developed all the time. Carriers and manufacturers retire (sunset) devices all the time. Different carriers might carry the same (or similar) device but might sunset (retire) the devices at a different time.

#### Tip

Developers should set a policy, made clear to users, of how long an application will be supported after the carrier or manufacturer stops supporting a specific device. This policy might need to be different for various carriers because carriers impose their own support requirements.

Developers need to understand how different kinds of device models can move through the worldwide marketplace. Some devices are available (or become popular) only in certain geographic regions. Sometimes devices are released worldwide, but often they are

released regionally. The T-Mobile G1, for example, was first released in the United States but later released worldwide. Similarly, the Motorola Droid was only available in the United States, whereas the similar Motorola Milestone was available only outside the United States.

Historically, it has been common for a device (or new generation of devices) to become available initially in market-driving regions of eastern Asia, including South Korea and Japan, and then show up in Europe, North America, and Australia, where device users often upgrade every year or two and pay premium rates for applications. Finally, these same devices become available in Central and South America, China, and India, where subscribers often don't have landlines or the same levels of income. Regions such as China and India must often be treated as entirely separate mobile marketplaces—with more affordable devices requiring vastly different revenue models. Here applications sell for less, but revenue is instead derived from the huge and growing subscriber base.

## Acquiring Target Devices

The earlier you can get your hands on the target devices, the better off you are. Sometimes this is as easy as going to the store and buying a new device. Other times, you need to acquire devices in other ways.

It is quite common for an application developer to target upcoming devices—devices not yet shipping or available to consumers. There is a great competitive advantage to having your application ready to run the moment consumers have the device in their hands for the first time. For preproduction devices, you can join manufacturer and operator developer programs. These programs help you keep abreast of changes to the device lines (upcoming models, discontinued models). Many of these programs also include pre-production device loan programs, enabling developers to get their hands on the device before consumers do.

### Tip

If you are just getting started acquiring Android devices, consider a Google Experience device, such as one of the Nexus handsets like the Galaxy Nexus. See <http://www.google.com/nexus/> for more information. As of this writing, phones targeted for just developers are no longer available.

There are risks for developers writing applications for specific preproduction devices because device shipment dates often slide and the platform might have unstable or bug-prone firmware. Devices are delayed or canceled. Device features (especially new and interesting ones) are not set in stone until the device ships and the developer verifies that those features work as expected. Exciting new devices are announced all the time—devices you might want your application to support. Your project plan must be flexible enough to change and adapt with the market as necessary.



### Tip

Sometimes you don't need to acquire specific devices to test with them. Various online services allow you to remotely install and test on real devices accessible and controllable through remote services. Most of these services have some sort of charge that must be weighed against the cost of actually owning the device outright.

## Determining the Feasibility of Application Requirements

Mobile developers are at the mercy of the device limitations, which vary in terms of memory and processing power, screen type, and platform version. Mobile developers do not really have the luxury traditional desktop application developers have of saying an application requires “more memory” or “more space.” Device limitations are pretty much fixed, and if a mobile application is to run, it runs within the device’s limitations, or not at all. Technically speaking, most Android devices have some hardware flexibility, such as the ability to use external storage devices such as SD cards, but we’re still talking about limited resources.

You can do true feasibility assessment only on the physical device, not the software emulator. Your application might work beautifully in the emulator but falter on the actual device. Mobile developers most constantly revisit feasibility, application responsiveness, and performance throughout the development process.

## Understanding Quality Assurance Risks

The quality assurance team has its work cut out for it because the testing environment is generally less than ideal.

### Testing Early, Testing Often

Get those target devices in hand as early as possible. For preproduction devices, it can take months to get the hardware in hand from the manufacturer. Cooperating with carrier device loaner programs and buying devices from retail locations is frustrating but sometimes necessary. Don’t wait until the last minute to gather the test hardware.

### Testing on the Device

It cannot be said enough: *Testing on the emulator is helpful, but testing on the device is essential.* In reality, it doesn’t matter if the application works on the emulator—no one uses an emulator in the real world.

Although you can perform factory resets on devices and wipe user data, there is often no easy way to completely “wipe” a device and return it to a clean starting state, so the quality assurance team needs to determine and stick to a testing policy of what is considered a clean state on the device. Testers might need to learn to flash devices with different firmware versions and understand subtle differences between platform versions, as well as how underlying application data is stored on the device (for example, SQLite databases, private application files, and cache usage).

## Mitigating the Risk of Limited Real-World Testing Opportunities

In some ways, every quality assurance tester works within a controlled environment. This is doubly true for mobile testers. They often work with devices not on real networks and preproduction devices that might not match those in the field. Add to this that because testing generally takes place in a lab, the location (including primary cell tower, satellite fixes and related device signal strength, availability of data services, LBS information, and locale information) is fixed. The quality assurance team needs to get creative to mitigate the risks of testing too narrow a range of these factors. For example, it is essential to test all applications when the device has no signal (and in airplane mode, and such) to make sure they don't crash and burn under such conditions that we all experience at some point in time. A variety of testing tools, some better suited to developers and white-box testers, are available to assist in application development. Some of the most suitable tools include Exerciser Monkey, monkeyrunner, and JUnit.

## Testing Client/Server and Cloud-Friendly Applications

Make sure the quality assurance team understands its responsibilities. Mobile applications often have network components and server-side functionality. Make sure thorough server and service testing is part of the overall test plan—not just the client portion of the overall solution that is implemented on the device. This might require the development of desktop or web applications to exercise network portions of the overall solution.

# Writing Essential Project Documentation

You might think that with its shorter schedules, smaller teams, and simpler functionality, mobile software project documentation would be less onerous. Unfortunately, this is not the case—quite the opposite. In addition to the traditional benefits any software project enjoys, good documentation serves a variety of purposes in mobile development.

Consider documenting the following for your project:

- Requirements analysis and prioritization
- Risk assessment and management
- Application architecture and design
- Feasibility studies, including performance benchmarking
- Technical specifications (overall, server, device-specific client)
- Detailed user-interface specifications (general, service-specific)
- Test plans, test scripts, test cases (general, device-specific)
- Scope change documentation

Much of this documentation is common in your average software development project. But perhaps your team finds that skimping on certain aspects of the documentation process has been doable in the past. Before you think of cutting corners in a mobile

development project, consider some of these documentation requirements for a successful project. Some project documentation might be simpler than that of larger-scale software projects, but other portions might need to be fleshed out in finer detail—especially user interface and feasibility studies.

## **Developing Test Plans for Quality Assurance Purposes**

Quality assurance relies heavily on the functional specification documentation and the user interface documentation. Screen real estate is valuable on the small screens of mobile devices, and user experience is vital to the successful mobile project. Test plans need to provide complete coverage of the application user interface, yet be flexible enough to address higher-level user experience issues that may meet the requirements of the test plan, but just aren't positive experiences.

### **Understanding the Importance of User Interface Documentation**

There's no such thing as a killer application with a poorly designed user interface. Thoughtful user interface design is one of the most important details to nail down during the design phase of any mobile software project. You must thoroughly document application workflow (application state) at the screen-by-screen level and can include detailed specifications for key usage patterns and how to gracefully fall back when certain keys or features are missing. You should clearly define usage cases in advance.

### **Leveraging Third-Party Testing Facilities**

Some companies opt to have quality assurance done offsite by a third party; most quality assurance teams require detailed documentation, including use case workflow diagrams to determine correct application behavior. If you do not provide adequate, detailed, and accurate documentation to the testing facility, you will not get deep, detailed, or accurate test results. By providing detailed documentation, you raise the bar from “it works” to “it works correctly.” What might seem straightforward to some people might not be to others.

### **Providing Documentation Required by Third Parties**

If you are required to submit your application to a software certification program or even, in some cases, to a mobile application store, part of your submission is likely to require some documentation about your application. Some stores require, for example, that your application include a Help feature or technical support contact information. Certification programs might require you to provide detailed documentation on application functionality, user interface workflow, and application state diagrams.

### **Providing Documentation for Maintenance and Porting**

Mobile applications are often ported to additional devices and other mobile platforms. This porting work is frequently done by a third party, making the existence of thorough functional and technical specifications even more crucial.

## Leveraging Configuration Management Systems

Many wonderful source control systems are out there for developers, and most that work well for traditional development work fine for a mobile project. Versioning your application, on the other hand, is not necessarily as straightforward as you might think.

### Choosing a Source Control System

Mobile development considerations impose no surprise requirements for source control systems. Some considerations for developers evaluating how they handle configuration management for a mobile project are

- Ability to keep track of source code (Java) and binaries (Android packages, and so on)
- Ability to keep track of application resources by device configuration (graphics and so on)
- Integration with the developer's chosen development environment (Eclipse)

One point to consider is integration between the development environment (such as Eclipse) and your source control system. Common source control systems such as Perforce, Subversion, and Mercurial work well with Eclipse. Check to see if your favorite source control works with your chosen Android development environment. Also, be aware that Eclipse is the recommended development environment for Android and that the Android development tools group is putting most of its effort into supporting Eclipse users. (See <http://tools.android.com/recent> for more information.)

### Implementing an Application Version System That Works

Developers should also make an early decision on a versioning scheme that takes into account the device particulars and the software build. It is often not sufficient to version the software by build alone (that is, Version 1.0.1).

Mobile developers often combine the traditional versioning scheme with the target device configuration or device class supported (Version 1.0.1.*Important Characteristic/Device Class Name*). This helps quality assurance, technical support personnel, and end users who might not know the model names or features of their devices or only know them by marketing names developers are often unaware of. For example, an application developed with camera support might be versioned 1.0.1.Cam, where Cam stands for “Camera Support,” whereas the same application for a device without camera support might have a version such as 1.0.1.NoCam, where NoCam stands for “No Camera Support” source branch. If you had two different maintenance engineers supporting the different source code trees, you would know just by the version name who to assign bugs to.

Just to make things a tad more confusing, you need to plan your upgrade versions as well. If an upgrade spawns a rebuild of your application, you might want to version it

appropriately: Version 1.0.1.NoCam.Upg1, and such. Yes, this can get out of control, so don't go overboard, but if you design your versioning system intelligently upfront, it can be useful later when you have different device builds floating around internally and with users. Finally, you also have to keep track of the `versionCode` attribute associated with your application.

Also, be aware of what distribution methods support multiple application packages or binaries as the same application and which require each binary to be managed independently. There are several good reasons to not have all of your code and resources in a single binary. Application package size, for example, can get large and unmanageable when the application attempts to support multiple device resolutions using alternative resources.

## Designing Mobile Applications

When designing an application for mobile, the developer must consider the constraints the device imposes and decide what type of application framework is best for a given project.

### Understanding Mobile Device Limitations

Applications are expected to be fast, responsive, and stable, but developers must work with limited resources. You must keep in mind the memory and processing power constraints of all target devices when designing and developing mobile applications.

### Exploring Common Mobile Application Architectures

Mobile applications have traditionally come in two basic models: standalone applications and network-driven applications.

Standalone applications are packaged with everything they require and rely on the device to do all the heavy lifting. All processing is done locally, in memory, and is subject to the limitations of the device. Standalone applications might use network functions, but they do not rely on them for core application functionality. An example of a reasonable standalone application is a basic Solitaire game. A user can play the game when the device is in Airplane Mode without issue.

Network-driven applications provide a lightweight client on the device but rely on the network (or “the cloud”) to provide a good portion of its content and functionality. Network-driven applications are often used to offload intensive processing to a server. Network-driven applications also benefit from the ability to deliver additional content or functionality on the fly, long after the application has been installed. Developers also like network-driven applications because this architecture enables them to build one smart application server or cloud service with device clients for many different operating systems to support a larger audience of users. Good examples of network-driven applications include

- Applications that leverage cloud-based services, application servers, or web services
- Customizable content such as ringtone and wallpaper applications
- Applications with noncritical process and memory-intensive operations that can be offloaded to a powerful server and the results delivered back to the client
- Any application that provides additional features at a later date without a full update to the binary

How much you rely on the network to assist in your application's functionality is up to you. You can use the network to provide only content updates (popular new ringtones), or you can use it to dictate how your application looks and behaves (for instance, adding new menu options or features on the fly).

## Designing for Extensibility and Maintenance

Applications can be written with a fixed user interface and a fixed feature set, but they need not be. Network-driven applications can be more complex to design but offer flexibility for the long term. Here's an example: Let's say you want to write a wallpaper application. Your application can be a standalone version, partially network driven, or completely network driven. Regardless, your application has two required functions:

- Display a set of images and allow the user to choose one.
- Take the chosen image and set it as the wallpaper on the device.

A super simple standalone wallpaper application might come with a fixed set of wallpapers. If they're a generic size for all target devices, you might need to reformat them for the specific device. You could write this application, but it would waste space and processing. You can't update the wallpapers available, and it is generally just a bad design.

The partially network-driven wallpaper application might enable the user to browse a fixed menu of wallpaper categories, which show images from a generic image server. The application downloads a specific graphic and then formats the image for the device. As the developer, you can add new wallpaper images to the server anytime, but you need to build a new application every time you want to add a new device configuration or screen size. If you want to change the menu to add live wallpapers at a later date, you need to write a new version of your application. This application design is feasible, but it isn't using its resources wisely either and isn't particularly extensible. However, you could use the single application server and write applications for Android, iPhone, BREW, J2ME, and Blackberry clients, so we are still in a better position than we were with the standalone wallpaper application.

The fully network-driven version of the wallpaper application does the bare minimum it needs to on the device. The client enables the server to dictate what the client user interface looks like, what menus to display, and where to display them. The user

browses the images from the application server just like with the partially network-driven version, but when the user chooses a wallpaper, the mobile application just sends a request to the server: “I want this wallpaper and I am this kind or type of device, with such and such screen resolution.” The server formats and resizes the image (any process-intensive operations) and sends the perfectly tailored wallpaper down to the application, which the application then sets as the wallpaper. Adding support for more devices is straightforward—simply deploy the lightweight client with any necessary changes and add support for that device configuration to the server. Adding a new menu item is just a server change, resulting in all devices (or whichever devices the server dictates) getting that new category. You only need to update the client when a new function requires the client to change, such as to add support for live wallpapers. The response time of this application depends on network performance, but the application is the most extensible and dynamic. However, this application is basically useless when the device is in Airplane Mode.

Standalone applications are straightforward. This approach is great for one-shot applications and those that are meant to be network independent. Network-driven applications require a bit more forethought and are sometimes more complicated to develop but might save a lot of time and provide users with fresh content and features for the long run.

## Designing for Application Interoperability

Mobile application designers should consider how they will interface with other applications on the device, including other applications written by the same developer. Some issues to address are

- Will your application rely on other content providers?
- Are these content providers guaranteed to be installed on the device?
- Will your application act as a content provider? What data will it provide?
- Will your application have background features? Act as a service?
- Will your application rely on third-party services or optional components?
- Will your application use publicly documented Intent mechanisms to access third-party functionality? Will your application provide the same?
- How will your application user experience suffer when optional components are not available?
- Will your application expose its functionality through a remote interface (AIDL)?

## Developing Mobile Applications

Mobile application implementation follows the same design principles as other platforms. The steps mobile developers take during implementation are fairly straightforward:

- Write and compile the code.
- Run the application in the software emulator.
- Test and debug the application in the software emulator or test device.
- Package and deploy the application to the target devices.
- Test and debug the application on the target devices.
- Incorporate changes from the team and repeat until the application is complete.



### Note

We talk more about development strategies for building solid Android applications in Chapter 17, “Designing and Developing Bulletproof Android Applications.”

## Testing Mobile Applications

Testers face many challenges, including device fragmentation (many devices, each with different features—some call this “compatibility”), defining device states (what is a clean state?), and handling real-world events (device calls, loss of coverage). Gathering the devices needed for testing can be costly and difficult.

The good news for mobile QA teams is that the Android SDK includes a number of useful tools for testing applications both on the emulator and the device. There are many opportunities for leveraging white-box testing.

You must modify defect-tracking systems to handle testing across device configurations and carriers. For thorough testing, QA team members generally cannot be given the device and told to “try to break it.” There are many shades of gray for testers, between black-box and white-box testing. Testers should know their way around the Android emulator and the other utilities provided with the Android SDK. Mobile quality assurance involves a lot of edge case testing. Again, a preproduction model of a device might not be exactly the same as what eventually ships to consumers.



### Note

We discuss testing Android applications in detail in Chapter 18, “Testing Android Applications.”

## Deploying Mobile Applications

Developers need to determine what methods they use to distribute applications. With Android, you have a number of options. You can market applications yourself and leverage third-party marketplaces such as Android Market. Consolidated mobile marketplaces, such as Handango, also have Android distribution channels of which you can take advantage.



### Note

We discuss publication of Android applications in detail in Chapter 19, “Publishing Your Android Application.”

## Determining Target Markets

Developers must take into account any requirements imposed by third parties offering application distribution mechanisms. Specific distributors might impose rules for what types of applications they distribute on your behalf. They might impose quality requirements such as testing certifications (although there are none specific to Android applications at the time this book went to print) and accompanying technical support, documentation and adherence to common user interface workflow standards (that is, the Back button should behave like so), and performance metrics for responsive applications. Distributors might also impose content restrictions such as barring objectionable content.



### Tip

The most popular distribution channels for Android applications have been changing over time. The Android Market remains the first stop in Android app publication, but the Amazon Appstore has gained a lot of traction in the past year. Other app stores are also available; some cater to special user groups and niche application genres, whereas others distribute for many different platforms. Check out our review of several popular markets on <http://goo.gl/IMuY2>.

## Supporting and Maintaining Mobile Applications

Developers cannot just develop an application, publish it, and forget about it—even the simplest of applications likely require some maintenance and the occasional upgrade. Generally speaking, mobile application support requirements are minimal if you come from a traditional software background, but they do exist.

Carriers and operators generally serve as the front line of technical support to end users. As a developer, you aren’t usually required to have 24/7 responsive technical support staff or toll-free device numbers and such. In fact, the bulk of application maintenance can fall on the server side and be limited to content maintenance—such as posting new media such as ringtones, wallpapers, videos, or other content. This may not seem obvious, at first. After all, you’ve provided your email and website when the user downloaded your application, right? Although that may be the case, your average user still calls the company on the bill first (in other words, the carrier) for support if the device is not functioning properly.

That said, the device firmware changes quickly, and mobile development teams need to stay on top of the market. Here are some of the maintenance and support considerations unique to mobile application development.

## Track and Address Crashes Reported by Users

The Android Market—the most popular way to distribute Android applications—has built-in features enabling users to submit crash and bug reports regarding your application. Monitor your developer account and address these issues in a timely fashion in order to maintain your credibility and keep your users happy.

## Testing Firmware Upgrades

Android handsets receive frequent (some say *too* frequent) firmware upgrades. This means that the Android platform versions you initially tested and supported become obsolete and the handsets your application is installed on can suddenly run new versions of the Android firmware. Although upgrades are supposed to be backward compatible, this hasn't always proven true. In fact, many developers have fallen victim to poor upgrade scenarios, in which their applications suddenly cease to function properly. Always retest your applications after a major or minor firmware upgrade occurs in the field.

## Maintaining Adequate Application Documentation

Maintenance is often not performed by the same engineers who developed the application in the first place. Here, keeping adequate development and testing documentation, including specifications and test scripts, is even more vital.

## Managing Live Server Changes

Always treat any live server and web or cloud service with the care it deserves. This means you need to appropriately time your backups and upgrades. You need to safeguard data and maintain user privacy at all times. You should manage rollouts carefully because live mobile application users might rely on the app's availability. Do not underestimate the server-side development or testing needs. Always test server rollouts and service upgrades in a safe testing environment before “going live.”

## Identifying Low-Risk Porting Opportunities

If you've implemented the device database we previously talked about in the chapter, now is the ideal time to analyze device similarities to identify easy porting projects. For example, you might discover the following: An application was originally developed for a specific class of device, but now several popular devices are on the market with similar specifications. Porting an existing application to these new devices is sometimes as straightforward as generating a new build (with appropriate versioning) and testing the application on the new devices. If you defined your device classes well, you might even get lucky and not have to make any changes at all when new devices come out.

## Summary

Mobile software development has evolved over time and differs in some important ways from traditional desktop software development. In this chapter, you gained some practical advice for adapting traditional software processes to mobile—from identifying target devices to testing and deploying your application to the world. There’s always room for improvement when it comes to software processes. Hopefully some of these insights can help you to avoid the pitfalls new mobile companies sometimes fall into or simply to improve the processes of veteran teams.

## References and More Information

Wikipedia on Software Process:

[http://en.wikipedia.org/wiki/Software\\_development\\_process](http://en.wikipedia.org/wiki/Software_development_process)

Wikipedia on Rapid Application Development (RAD):

[http://en.wikipedia.org/wiki/Rapid\\_application\\_development](http://en.wikipedia.org/wiki/Rapid_application_development)

Wikipedia on Iterative Development:

[http://en.wikipedia.org/wiki/Iterative\\_and\\_incremental\\_development](http://en.wikipedia.org/wiki/Iterative_and_incremental_development)

Wikipedia on Waterfall Development Process:

[http://en.wikipedia.org/wiki/Waterfall\\_model](http://en.wikipedia.org/wiki/Waterfall_model)

Extreme Programming:

<http://www.extremeprogramming.org>

# Designing and Developing Bulletproof Android Applications

In this chapter, we cover tips and techniques from our years in the trenches of mobile software design and development. We also warn you—the designers, developers, and managers of mobile applications—of the various and sundry pitfalls you should do your best to avoid. Reading this chapter all at one time when you’re new to mobile development might be a bit overwhelming. Instead, consider reading specific sections when planning the parts of the overall process. Some of our advice might not be appropriate for your particular project, and processes can always be improved. Hopefully this information about how mobile development projects succeed (or fail) gives you some insight into how you might improve the chances of success for your own projects.

## Best Practices in Designing Bulletproof Mobile Applications

The “rules” of mobile application design are straightforward and apply across all platforms. These rules were crafted to remind us that our applications often play a secondary role on the device. Many Android devices are, at the end of the day, [smart]phones first. These rules also make it clear that we do operate, to some extent, because of the infrastructure managed by the carriers and device manufacturers. These rules are echoed throughout the Android Software Development Kit (SDK) License Agreement and those of third-party application marketplace terms and conditions.

These “rules” are as follows:

- Don’t interfere with device telephony and messaging services (if applicable).
- Don’t break or otherwise tamper with or exploit the device hardware, firmware, software, or OEM components.
- Don’t abuse or cause problems on operator networks.
- Don’t abuse the user’s trust.

Now perhaps these rules sound like no-brainers, but even the most well-intentioned developers can accidentally fall into some of these categories if they aren't careful and don't test their applications thoroughly before distribution. This is especially true for applications that leverage networking support and low-level hardware APIs on the device, and those that store private user data such as names, locations, and contact information.

## Meeting Mobile Users' Demands

Mobile users also have their own set of demands for applications they install on their devices. Applications are expected to

- Be responsive, stable, and secure
- Have straightforward, intuitive user interfaces that are easy to get up and running
- Get the job done with minimal frustration to the user (responsive and familiar) and minimal impact on device performance (battery usage, network and data usage, and so on)
- Be available 24 hours a day, 7 days a week (remote servers or services always on, always available, not running in someone's closet)
- Include a Help and/or About screen for feedback and support contact information
- Honor private user information and treat it with care

## Designing User Interfaces for Mobile Devices

Designing effective user interfaces for mobile devices, especially applications that run on a number of different devices, is something of a black art. We've all seen bad mobile application user interfaces. A frustrating user experience can turn a user off your brand; a good experience can win a user's loyalty. Great experiences give your application an edge over the competition, even if your functionality is similar. An elegant, well-designed user interface can win over users even when the application functionality is behind the competition. Said another way, doing something really well is more important than cramming too many features into an app and doing them badly.

Here are some tips for designing great mobile user interfaces:

- Fill screens sparingly; too much information on one screen overwhelms the user.
- Be consistent with user interface workflows, menu types, and buttons. Also, consider the device norms with this consistency.
- Design your applications using fragments, even if you aren't targeting tablet devices. (The Android Support Package makes this possible for nearly all target versions.)
- Make Touch Mode "hit areas" large enough and spaced appropriately.
- Streamline common use cases with clear, consistent, and straightforward interfaces.

- Use big, readable fonts and large icons.
- Integrate tightly with other applications on the system using standardized controls, such as the quick Contact badge, content providers, and search adapters.
- Keep localization in mind when designing text-heavy user interfaces. Some languages are lengthier than others.
- Reduce keys or clicks needed as much as possible.
- Do not assume that specific input mechanisms (such as specific buttons or the existence of a keyboard) are available on all devices.
- Try to design the default use case of each screen to require only the user's thumb. Special cases might require other buttons or input methods, but encourage "thumbing" by default.
- Size resources such as graphics appropriately for target devices. Do not include oversized resources and assets because they bulk up your application package, load more slowly, and are generally less efficient.
- In terms of "friendly" user interfaces, assume that users do not read the application permissions when they approve them to install your application. If your application does anything that could cause the user to incur significant fees or shares private information, consider informing them again (as appropriate) when your application performs such actions. Basically, take a "no surprises" approach, even if the permissions and your privacy policy also state the same thing.



### Note

We discuss how to design Android applications that are compatible with a wide range of devices, including how to develop for different screen sizes and resolutions, in Chapter 15, "Designing Compatible Applications."

## Designing Stable and Responsive Mobile Applications

Mobile device hardware has come a long way in the past few years, but developers must still work with limited resources. Users do not usually have the luxury of upgrading the RAM and other hardware in Android devices. Android users can, however, take advantage of removable storage devices such as SD cards to provide some "extra" space for application and media storage. Spending some time upfront to design a stable and responsive application is important for the success of the project. The following are some tips for designing robust and responsive mobile applications:

- Don't perform resource-intensive or lengthy operations on the main UI thread. Always use asynchronous tasks, threads, or background services to offload blocking operations.

- Use efficient data structures and algorithms; these choices manifest themselves in app responsiveness and happy users.
- Use recursion with care; these functional areas should be code reviewed and performance tested.
- Keep application state at all times. The Android activity stack makes this work well, but you should take extra care to go above and beyond.
- Save your state using appropriate lifecycle callbacks and assume that your application will be suspended or stopped at any moment. If your application is suspended or closed, you cannot expect a user to verify anything (click a button, and so on). If your application resumes gracefully, your users will be grateful.
- Start up fast and resume fast. You cannot afford to have the user twiddling thumbs waiting for your application to start. Instead, you need to strike a delicate balance between preloading and on-demand data because your application might be suspended (or closed) with no notice.
- Inform users during long operations by using progress bars. Consider offloading heavy processing to a server instead of performing these operations on the device because these operations might drain battery life beyond the limits users are willing to accept.
- Ensure that long operations are likely to succeed before embarking on them. For example, if your application downloads large files, check for network connectivity, file size, and available space before attempting the download.
- Minimize the use of local storage, because most devices have very limited amounts. Use external storage, when appropriate. Be aware that SD cards (the most common external storage option) can be ejected and swapped; your application should handle this gracefully.
- Understand that data calls to content providers and across the AIDL barrier come at a cost to performance, so make these calls judiciously.
- Verify that your application resource consumption model matches your target audience. Gamers might anticipate shorter battery life on graphics-intensive games, but productivity applications should not drain the battery unnecessarily and should be lightweight for people “on the go” who do not always have their device charging.



### Tip

Written by the Google Android team, the Android Developers blog (<http://android-developers.blogspot.com>) is a fantastic resource. This blog provides detailed insight into the Android platform, often covering topics not discussed in the Android platform documentation. Here you can find tips, tricks, best practices, and shortcuts on relevant Android development topics such as memory management (such as Context management), view optimization (avoiding deep view hierarchies), and layout tricks to improve UI speed.

Savvy Android developers visit this blog regularly and incorporate these practices and tips into their projects. Keep in mind that Google's Android Developer guys and gals are often focused on educating the rest of us about the latest API-level features; their techniques and advice may not always be suitable for implementation with older target platforms.

## Designing Secure Mobile Applications

Many mobile applications integrate with core applications such as the Phone, Camera, and Contacts. Make sure you take all the precautions necessary to secure and protect private user data such as names, locations, and contact information used by your application. This includes safeguarding personal user data on application servers and during network transmission.

### Tip

If your application accesses, uses, or transmits private data, especially usernames, passwords, or contact information, it's a good idea to include an End User License Agreement (EULA) and a privacy policy with your application. Also keep in mind that privacy laws vary by country.



## Handling Private Data

To begin with, limit the private or sensitive data your application stores as much as possible. Don't store this information in plain text, and don't transmit it over the network without safeguards. Do not try to work around any security mechanisms imposed by the Android framework. Store private user data in private application files, which are private to the application, and not in shared parts of the operating system. Do not expose application data in content providers without enforcing appropriate permissions on other applications. Use the encryption classes available in the Android framework when necessary.

## Transmitting Private Data

The same cautions should apply to any remote network data storage (such as application servers or cloud storage) and network transmission. Make sure any servers or services that your application relies on are properly secured against identity or data theft and invasion of privacy. Treat any servers your application uses like any other part of the application—test these areas thoroughly. Any private data transmitted should be secured using typical security mechanisms such as SSL. The same rules apply when enabling your application for backups using services such as Android Backup Service.

## Designing Mobile Applications for Maximum Profit

For billing and revenue generation, mobile applications generally fall into one of four categories:

- Free applications (including those with advertising revenue)
- Single payment (pay once)
- In-application payment for content (pay for specific content, such as a ringtone, a Sword of Smiting, or a new level pack)
- Subscription payments (pay on a schedule, often seen with productivity and service applications)

Applications can use multiple types of billing, depending on which marketplaces and billing APIs they use (Android Market, for example, limits billing methods to Google Checkout). No specific billing APIs are built in to the Android framework. With Android in general, third parties can provide billing methods or APIs, so technically the sky's the limit. That said, Google provides an optional In-App Billing API add-on for use with the Android Market (and only the Android Market).

When designing your mobile applications, consider the functional areas where billing can come into play and factor this into your design. Consider the transactional integrity of specific workflow areas of the application that can be charged for. For example, if your application has the capability to deliver data to the device, make sure this process is transactional in nature so that if you decide to charge for this feature, you can drop in the billing code, and when the user pays, the delivery occurs, or the entire transaction is rolled back.



### Note

You learn more about the different methods currently available to market your application in Chapter 19, “Publishing Your Android Application” and more about the Android Market’s in-app billing API in *Android Wireless Application Development Volume II: Advanced Topics*.

## Leveraging Third-Party Quality Standards

There are no certification programs specifically designed for Android applications. However, as more applications are developed, third-party standards might be designed to differentiate quality applications from the masses. For example, mobile marketplaces may impose quality requirements, and certainly programs have been created with some recognized body’s endorsement or stamp of approval. The Amazon Appstore for Android puts apps through some testing before they are made available for sale. The Android Market has an Editor’s Choice category of applications. Developers with an eye on financial applications would do well to consider conformance requirements.



### Warning

With Android, the market is expected to manage itself to a greater extent than some other mobile platform markets. Do not make the mistake of interpreting that as “no rules” when it really means “few rules imposed by the system.” Strong licensing terms are in place to keep malware and other malicious code out of users’ hands, and applications do indeed get removed for misbehavior, just as they do when they sneak through onto other platform markets.

## Designing Mobile Applications for Ease of Maintenance and Upgrades

Generally speaking, it's best to make as few assumptions about the device configurations as possible when developing a mobile application. You'll be rewarded later when you want to port your application or provide an easy upgrade. You should carefully consider what assumptions you make.

### Leveraging Application Diagnostics

In addition to adequate documentation and easy-to-decipher code, you can leverage some tricks to help maintain and monitor mobile applications in the field. Building lightweight auditing, logging, and reporting into your application can be highly useful for generating your own statistics and analytics. Relying on third-party information, such as that generated with market reports, could cause you to miss some key pieces of data that are useful to you. For example, you can easily keep track of

- How many users install the application
- How many users launch the application for the first time
- How many users regularly use the application
- What the most popular usage patterns and trends are
- What the least popular usage patterns and features are
- What devices (determined by application versioning or other relevant metrics) are the most popular

Often you can translate these figures into rough estimates of expected sales, which you can later compare with actual sales figures from third-party marketplaces. You can streamline and, for example, make the most popular usage patterns the most visible and efficient in terms of user experience design. Sometimes you can even identify potential bugs, such as features that are not working at all, just by noting that a feature has never been used in the field. Finally, you can determine which device targets are most appropriate for your specific application and user base.

You can gather interesting information about your application from numerous sources, including the following:

- Android Market sales statistics, ratings, and bug/crash reports, as well as those available on other distribution channels.
- Application integration with statistics-gathering APIs such as Google Analytics (discussed in detail in *Android Wireless Application Development Volume II: Advanced Topics* of this book series).
- For applications relying on network servers, quite a lot of information can be determined by looking at server-side statistics.
- Feedback sent directly to you, the developer, through email or other mechanisms made available to your users.



### Tip

Never collect personal data without the user's knowledge and consent. Gathering anonymous diagnostics is fairly commonplace, but avoid keeping any data that can be considered private. Make sure your sample sizes are large enough to obfuscate any personal user details, and make sure to factor out any live QA testing data from your results (especially when considering sales figures).

## Designing for Easy Updates and Upgrades

Android applications can easily be upgraded in the field. The application update and upgrade processes do pose some challenges to developers, though. By *updating*, we're talking about modifying the Android manifest version information and redeploying the updated application on users' devices. By *upgrading*, we're talking about creating an entirely new application package with new features and deploying it as a separate application that the user needs to choose to install and that does not replace the old application.

From an update perspective, you need to consider what conditions necessitate an update in the field. For example, do you draw the line at crashes or feature requests? You also want to consider the frequency with which you deploy updates—you need to schedule updates such that they come up frequently enough to be truly useful, but not so often that the users are constantly updating their application.



### Tip

You should build application content updates into the application functionality as a feature (often network driven) as opposed to necessitating an over-the-air actual application update. By enabling your applications to retrieve fresh content on the fly, you keep your users happy longer and applications stay relevant.

When considering upgrades, decide the manner in which you will migrate users from one version of your application to the next. Will you leverage Android backup features so that your users can transition seamlessly from one device to the next, or will you provide your own solution? Consider how you will inform users of existing applications that a major new version is available.

## Leveraging Android Tools for Application Design

The Android SDK and developer community provide a number of useful tools and resources for application design. You might want to leverage the following tools during this phase of your development project:

- The Android emulator is a good place to start for rapid proof of concept, before you have specific devices. You can use different Android Virtual Device (AVD) configurations to simulate different device configurations and platform versions.
- The DDMS tool is very useful for memory profiling.

- The Hierarchy Viewer in Pixel Perfect View enables accurate user interface design. Along with layoutopt, it can also be used to optimize your layout designs.
- The Draw Nine-Patch tool can create stretchable graphics for mobile use.
- Real devices may be your most important tool. Use real devices for feasibility research and application proof-of-concept work, whenever possible. Do not design solely using the emulator.
- The technical specifications for specific devices, often available from manufacturers and carriers, can be invaluable for determining the configuration details of target devices.

## Avoiding Silly Mistakes in Android Application Design

Last but not least, here is a list of some of the silly mistakes Android designers should do their best to steer clear of:

- Designing or developing for months without performing feasibility testing on the device (basically “waterfall testing”)
- Designing for a single device, platform, language, or hardware
- Designing as if your device has a large amount of storage and processing power and is always plugged in to a power source
- Developing for the wrong version of the Android SDK (verify device SDK version)
- Trying to adapt applications to smaller screens after the fact by having the device “scale”
- Deploying oversized graphics and media assets with an application instead of sizing them appropriately

## Best Practices in Developing Bulletproof Mobile Applications

Developing applications for mobile is not that different from traditional desktop development. However, developers might find developing mobile applications more restrictive, especially resource constrained. Again, let's start with some best practices or “rules” for mobile application development:

- Test assumptions regarding feasibility early and often on the target devices.
- Keep application size as small and efficient as possible.
- Choose efficient data structures and algorithms appropriate to mobile.

- Exercise prudent memory management.
- Assume that devices are running primarily on battery power.

## Designing a Development Process That Works for Mobile Development

A successful project's backbone is a good software process. It ensures standards, good communication, and reduces risks. We talked about the overall mobile development process in Chapter 16, "The Android Software Development Process." Again, here are a few general tips of successful mobile development processes:

- Use an iterative development process.
- Use a regular, reproducible build process with adequate versioning.
- Communicate scope changes to all parties—changes often affect testing most of all.

## Testing the Feasibility of Your Application Early and Often

It cannot be said enough: You must test developer assumptions on real devices. There is nothing worse than designing and developing an application for a few months only to find that it needs serious redesign to work on an actual device. Just because your application works on the emulator does not, *in any way*, guarantee that it will run properly on the device. Some functional areas to examine carefully for feasibility include

- Functionality that interacts with peripherals and device hardware
- Network speed and latency
- Memory footprint and usage
- Algorithm efficiency
- User interface suitability for different screen sizes and resolutions
- Device input method assumptions
- File size and storage usage

We know—we sound like a broken record but, truly, we've seen this mistake happen over and over again. Projects are especially vulnerable to this when target devices aren't yet available. What happens is that engineers are forced closer to the waterfall method of software development with a big, bad surprise after weeks or months of development on some vanilla-style emulator.

We don't need to explain again why waterfall approaches are dangerous, do we? You can never be too cautious about this stuff. Think of this as the preflight safety speech of mobile software development.

## Using Coding Standards, Reviews, and Unit Tests to Improve Code Quality

Developers who spend the time and effort necessary to develop efficient mobile application are rewarded by their users. The following is a representative list of some of the efforts you can take:

- Centralizing core features in shared Java packages. (If you have shared C or C++ libraries, consider using the Android NDK.)
- Developing for compatible versions of the Android SDK (know your target devices).
- Using the right level of optimization, including coding with RenderScript or using the NDK, where appropriate. These topics are discussed in *Android Wireless Application Development Volume II: Advanced Topics*.
- Using built-in controls and widgets appropriate to the application, customizing only where needed.

You can use system services to determine important device characteristics (screen type, language, date, time, input methods, available hardware, and so on). If you make any changes to system settings from within your application, be sure to change the settings back when your application exits or pauses, if appropriate.

### Defining Coding Standards

Developing a set of well-communicated coding standards for the development team can help drive home some of the important requirements of mobile applications. Some standards might include

- Implementing robust error handling as well as handling exceptions gracefully.
- Moving lengthy, process-intensive, or blocking operations off the main UI thread.
- Avoiding creating unnecessary objects during critical sections of code or user interface behavior, such as animations and response to user input.
- Releasing objects and resources you aren't actively using.
- Practicing prudent memory management. Memory leaks can render your application useless.
- Using resources appropriately for future localization. Don't hardcode strings and other assets in code or layout files.
- Avoiding obfuscation in the code itself unless you're doing so for a specific reason (such as using Google's License Verification Library, or LVL). Comments are worthwhile. However, you should consider obfuscation later in the development process to protect against software piracy using built-in ProGuard support.
- Considering using standard document generation tools such as Javadoc.
- Instituting and enforce naming conventions—in code and in database schema design.

## Performing Code Reviews

Performing code inspections can improve the quality of project code, help enforce coding standards, and identify problems before QA gets their hands on a build and spends time and resources testing it.

It can also be helpful to pair developers with the QA tester who tests their specific functional areas to build a closer relationship between the teams. If testers understand how the application and Android operating system function, they can test the application more thoroughly and successfully. This might or might not be done as part of a formal code review process. For example, a tester can identify defects related to type-safety just by noting the type of input expected (but not validated) on a form field of a layout or by reviewing the Submit or Save button handling function with the developer. This would help circumvent the time spent to file, review, fix, and retest validation defects. Reviewing the code in advance doesn't reduce the testing burden, but rather helps reduce the number of easily caught defects.

## Developing Code Diagnostics

The Android SDK provides a number of packages related to code diagnostics. Building a framework for logging, unit testing, and exercising your application to gather important diagnostic information, such as the frequency of method calls and performance of algorithms, can help you develop a solid, efficient, and effective mobile application. It should be noted that diagnostic hooks are almost always removed prior to application publication because they impose significant performance reductions and greatly reduce responsiveness.

## Using Application Logging

In Chapter 3, “Writing Your First Android Application,” we discuss how to leverage the built-in logging class `android.util.Log` to implement diagnostic logging, which can be monitored via a number of Android tools, such as the LogCat utility (available within DDMS, ADB, and the Android Development Plug-in for Eclipse).

## Developing Unit Tests

Unit testing can help developers move one step closer to the elusive 100% of code coverage testing. The Android SDK includes extensions to the JUnit framework for testing Android applications. Automated testing is accomplished by creating test cases, in Java code, that verify that the application works the way you designed it. You can do this automated testing for both unit testing and functional testing, including user interface testing.

Basic JUnit support is provided through the `junit.framework` and `junit.runner` packages. Here you find the familiar framework for running basic unit tests with helper classes for individual test cases. You can combine these test cases into test suites. There are utility classes for your standard assertions and test result logic.

The Android-specific unit testing classes are part of the `android.test` package, which includes an extensive array of testing tools designed specifically for Android applications. This package builds upon the JUnit framework and adds many interesting features, such as the following:

- Simplified Hooking of Test Instrumentation (`android.app.Instrumentation`) with `android.test.InstrumentationTestCase`, which you can run via ADB shell commands
- Performance Testing (`android.test.PerformanceTestCase`)
- Single Activity (or Context) Testing (`android.test.ActivityUnitTestCase`)
- Full Application Testing (`android.test.ApplicationTestCase`)
- Services Testing (`android.test.ServiceTestCase`)
- Utilities for generating events such as touch events (`android.test.TouchUtils`)
- Many more specialized assertions (`android.test.MoreAsserts`)
- View validation (`android.test.ViewAsserts`)

#### Tip

If you are interested in designing and implementing a unit test framework for your Android application, we suggest working through our tutorial called “Android SDK: Unit Testing with the JUnit Testing Framework.” You can find this tutorial online on our blog at <http://androidbook.blogspot.com/2010/08/unit-testing-with-android-junit.html>.



## Handling Defects Occurring on a Single Device

Occasionally, you have a situation in which you need to provide code for a specific device. Google and the Android team tell you that when this happens, it’s a bug, so you should tell them about it. By all means, do so. However, this won’t help you in the short term. Nor will it help you if they fix it in a subsequent revision of the platform but carriers don’t roll out the update and fix for months, if ever, to specific devices.

Handling bugs that occur only on a single device can be tricky. You don’t want to branch code unnecessarily, so here are some of your choices:

- If possible, keep the client generic and use the server to serve up device-specific items.
- If the conditions can be determined programmatically on the client, try to craft a generic solution that enables developers to continue to develop under one source code tree, without branching.
- If the device is not a high-priority target, consider dropping it from your requirements if the cost-benefit ratio suggests that a workaround is not cost effective. Not all markets support excluding individual devices, but Android Market does.

- If required, branch the code to implement the fix. Make sure to set your Android manifest file settings such that the branched application version is installed only on the appropriate devices.
- If all else fails, document the problem only and wait for the underlying “bug” to be addressed. Keep your users in the loop.

## Leveraging Android Tools for Development

The Android SDK comes with a number of useful tools and resources for application development. The development community adds even more useful utilities to the mix. You might want to leverage the following tools during this phase of your development project:

- The Eclipse development environment with the ADT plug-in
- The Android emulator and physical devices for testing
- The Android Dalvik Debug Monitor Service (DDMS) tool for debugging and interaction with the emulator or device
- The Android Debug Bridge (ADB) tool for logging, debugging, and shell access tools
- The `sqlite3` command-line tool for application database access (available via the ADB shell)
- The Hierarchy Viewer for user interface debugging of views

Numerous other tools also are available as part of the Android SDK. See the Android documentation for more details.

## Avoiding Silly Mistakes in Android Application Development

Here are some of the frustrating and silly mistakes Android developers should try to avoid:

- Forgetting to register new activities, services, and necessary permissions to the `AndroidManifest.xml` file
- Forgetting to display `Toast` messages using the `show()` method
- Hard-coding information such as network information, test user information, and other data into the application
- Forgetting to disable diagnostic logging before release
- Distributing live applications with debug mode enabled

## Summary

Be responsive, stable, and secure—these are the tenets of Android development. In this chapter, we armed you—the software designers, developers, and project managers—with tips, tricks, and best practices for mobile application design and development based on real-world knowledge and experience from veteran mobile developers. Feel free to pick and choose which information works well for your specific project, and keep in mind that the software process, especially the mobile software process, is always open to improvement.

## References and More Information

Android Best Practices: Designing for Performance:

<http://developer.android.com/guide/practices/design/performance.html>

Android Best Practices: Designing for Responsiveness:

<http://developer.android.com/guide/practices/design/responsiveness.html>

Android Best Practices: Designing for Seamlessness:

<http://developer.android.com/guide/practices/design/seamlessness.html>

Android Best Practices: User Interface Guidelines:

[http://developer.android.com/guide/practices/ui\\_guidelines/index.html](http://developer.android.com/guide/practices/ui_guidelines/index.html)

Getting Started with Google Analytics on Android:

<http://www.devx.com/wireless/Article/47049>

Android App Publication: A Checklist of Pre-Publication Considerations:

<http://mobile.tutsplus.com/tutorials/android/android-app-publication-a-checklist-of-pre-publication-considerations/> (<http://goo.gl/GDhpz>)

Where to Sell Your Killer App:

<http://www.developer.com/ws/where-to-sell-your-android-killer-app.html>

*This page intentionally left blank*

# Testing Android Applications

Test early, test often, test on the device. That is the quality assurance mantra we consider most important when it comes to testing Android applications. Testing your applications need not be an onerous process. Instead, you can adapt traditional quality assurance techniques, such as automation and unit testing to the Android platform with relative ease. In this chapter, we discuss our tips and tricks for testing Android applications. We also warn you—the project managers, software developers, and testers of mobile applications—of the various and sundry pitfalls you should do your best to avoid.

## Best Practices in Testing Mobile Applications

Like all quality assurance processes, mobile development projects benefit from a well-designed defect tracking system, regularly scheduled builds, and planned, systematic testing. There are also plentiful opportunities for white-box and black-box testing as well as opportunities for automation.

### Designing a Mobile Application Defect Tracking System

You can customize most defect tracking systems to work for the testing of mobile applications. The defect tracking system must encompass tracking of issues for specific device defects and problems related to any centralized application servers (if applicable).

### Logging Important Defect Information

A good mobile defect tracking system includes the following information about a typical device defect:

- Application build version information, language, and so on.
- Device configuration and state information, including device type, Android platform version, and important specs.
- Screen orientation, network state, sensor information.
- Steps to reproduce the problem using specific details about exactly which input methods were used (touch versus click).

- Device screenshots that can be taken using DDMS or the Hierarchy Viewer tool provided with the Android SDK.



### Tip

It can be helpful to develop a simple glossary of standardized terms for certain actions on the devices, such as touch mode gestures, click versus tap, long-click versus press-and-hold, clear versus back, and so on. This helps make the steps to reproduce a defect more precise to all parties involved.

### Redefining the Term Defect for Mobile Applications

It's also important to consider the larger definition of the term *defect*. Defects might occur on all devices or on only some devices. Defects might also occur in other parts of the application environment, such as on a remote application server. Some types of defects typical on mobile applications include

- Crashing, unexpected terminations, force closures, app not responding (ANR) events, and various other terms used for unexpected behavior that results in the application no longer running or responding.
- Features not functioning correctly (improper implementation).
- Using too much disk space on the device.
- Inadequate input validation (typically, button mashing).
- State management problems (startup, shutdown, suspend, resume, power off).
- Responsiveness problems (slow startup, shutdown, suspend, resume).
- Inadequate state change testing (failures during interstate changes, such as an unexpected interruption during resume).
- Usability issues related to input methods, font sizes, and cluttered screen real estate. Cosmetic problems that cause the screen to display incorrectly.
- Pausing or “freezing” on the main UI thread (failure to implement asynchronous tasks, threading).
- Feedback indicators missing (failure to indicate progress).
- Integration with other applications on the device causing problems.
- Application “not playing nicely” on the device (draining battery, disabling power-saving mode, overusing networking resources, incurring extensive user charges, obnoxious notifications).
- Using too much memory, not freeing memory or releasing resources appropriately, and not stopping worker threads when tasks are finished.
- Not conforming to third-party agreements, such as Android SDK License Agreement, Google Maps API terms, marketplace terms, or any other terms that apply to the application.

- Application client or server not handling protected/private data securely. This includes ensuring that remote servers or services have adequate uptime and security measures taken.

## Managing the Testing Environment

Testing mobile applications poses a unique challenge to the QA team, especially in terms of configuration management. The difficulty of such testing is often underestimated. Don't make the mistake of thinking that mobile applications are easier to test because they have fewer features than desktop applications and are, therefore, simpler to validate. The vast variety of Android devices available on the market today makes testing different installation environments tricky.

### Warning

Ensure that all changes in project scope are reviewed by the quality assurance team. Adding new devices sometimes has little impact on the development schedule but can have significant consequences in terms of testing schedules.

## Managing Device Configurations

Device fragmentation is one of the biggest challenges the mobile tester faces. Android devices come in various form-factors with different screens, platform versions, and underlying hardware. They come with a variety of input methods such as hardware buttons, keyboards, and touch screens. They come with optional features, such as cameras, enhanced graphics support, fingerprint readers, and even 3D displays. Many Android devices are smartphones, but non-phone devices such as Android tablets, TVs, and other devices are becoming more and more popular with each Android SDK release. Keeping track of all the devices, their abilities, and so on is a big job, and much of the work falls on the test team.

QA personnel must have a detailed understanding of the functionality available of each target device, including familiarity with what features are available and any device-specific idiosyncrasies that exist. Whenever possible, testers should test each device as it is used in the field, which might not be the device's default configuration or language. This means changing input modes, screen orientations, and locale settings. It also means testing with battery power, not just plugging it in to a power source while sitting at a desk.

### Tip

Be aware of how third-party firmware modifications can affect how your application works on the device. For example, let's assume you've gotten your hands on an unbranded version of a target device and testing has gone well. However, if certain carriers take that same device, but remove some default applications and load it up with others, this is valuable information to the tester. Many devices now ditch the stock Android user experience for more custom user interfaces, like Motorola's Motoblur, HTC's Sense, and Samsung's

TouchWiz user interfaces. Just because your application runs flawlessly on the “vanilla” device doesn’t mean that this is how most users’ devices are configured by default. Do your best to get test devices that closely resemble the devices users will have in the field. The various default styles may not display as you expect with your user interface.

One hundred percent testing coverage is impossible, so QA must develop priorities thoughtfully. As we discuss in Chapter 16, “The Android Software Development Process,” developing a device database can greatly reduce the confusion of mobile configuration management, help determine testing priorities, and keep track of physical hardware available for testing. Using AVD configurations, the emulator is also an effective tool for extending coverage to simulate devices and situations that would not be covered otherwise.



### Tip

If you have trouble configuring devices for real-life situations, you might want to look into the device “labs” available through some carriers. Instead of using loaner programs, developers visit the carrier’s onsite lab where they can rent time on specific devices. Here, a developer can install an application and test it—not ideal for recurring testing but much better than no testing, and some labs are staffed with experts to help out with device-specific issues.

## Determining Clean Starting State on a Device

There is currently no good way to “image” a device so that you can return to the same starting state again and again. The QA test team needs to define what a “clean” device is for the purposes of test cases. This can involve a specific uninstall process, some manual cleanup, or sometimes a factory reset.



### Tip

Using the Android SDK tools, such as DDMS and ADB, developers and testers can have access to the Android file system, including application SQLite databases. These tools can be used to monitor and manipulate data on the emulator. For example, testers might use the `sqlite3` command-line interface to “wipe” an application database or fill it with test data for specific test scenarios. For use on devices, you may need to “root” the devices first. Rooting a device is beyond the scope of this book, and we do not recommend doing so on test devices.

While we’re on the topic of “clean” states, here’s another issue to consider: You may have heard that you can “root” most Android devices, allowing access to underlying device features not openly accessible through the public Android SDK. Certainly there are apps (and developers writing apps) that require this kind of access (some are even published on Android Market). Generally speaking, though, we feel that rooted devices do not make good testing and development devices for most teams. You want to be developing

and testing on devices that resemble those in the hands of users; most users do not root their devices.

### Mimicking Real-World Activities

It is nearly impossible (and certainly not cost-effective for most companies) to set up a complete isolated environment for mobile application testing. It's fairly common for networked applications to be tested against test (mock) application servers and then go "live" on production servers with similar configurations. However, in terms of device configuration, mobile software testers must use real devices with real service to test mobile applications properly. If the device is a phone, then it needs to be able to make and receive phone calls, send and receive text messages, determine location using LBS services, and basically do anything a phone would normally do.

Testing a mobile application involves more than just making sure the application works properly. In the real world, your application does not exist in a vacuum but is one of many installed on the device. Testing a mobile application involves ensuring that the software integrates well with other device functions and applications. For example, let's say you were developing a game. Testers must verify that calls received while playing the game cause the game to automatically pause (keep state) and allow calls to be answered or ignored without issue.

This also means testers must install other applications on to the device. A good place to start is with the most popular applications for the device. Testing your application not only with these applications installed, but also with real use, can reveal integration issues or usage patterns that don't mesh well with the rest of the device.

Sometimes testers need to be creative when it comes to reproducing certain types of events. For example, testers must ensure that their application behaves appropriately when mobile handsets lose network connectivity or coverage.



#### Tip

Unlike with some other mobile platforms, testers actually have to take special steps to make most Android devices lose coverage above and beyond holding them wrong. To test loss of signal, you could go out and test your application in a highway tunnel or elevator, or you could just place the device in the refrigerator. Don't leave it in the cold too long, though, because this will drain the battery. Tin cans work great, too, especially those that have cookies in them: First, eat the cookies; then place the device in the can to seal off the signal. This advice also holds true for testing applications that leverage location-based services.

### Maximizing Testing Coverage

All test teams strive for 100% testing coverage, but most also realize such a goal is not reasonable or cost effective (especially with dozens of Android devices available around the world). Testers must do their best to cover a wide range of scenarios, the depth and breadth of which can be daunting—especially for those new to mobile. Let's look at

several specific types of testing and how QA teams have found ways—some tried-and-true and others new and innovative—to maximize coverage.

### **Validating Builds and Designing Smoke Tests**

In addition to a regular build process, it can be helpful to institute a build acceptance test policy (also sometimes called build validation, smoke testing, or sanity testing). Build acceptance tests are short and targeted at key functionality to determine whether the build is good enough for more thorough testing to be completed. This is also an opportunity to quickly verify bug fixes expected to be in the build before a complete retesting cycle occurs. Consider developing build-acceptance tests for multiple Android platform versions to run simultaneously.

### **Automating Testing**

Mobile build acceptance testing is frequently done manually on the highest-priority target device; however, this is also an ideal situation for an automated “sanity” test. By creating an automated test script that runs using the Android SDK’s test tool, called monkeyrunner, the team can increase its level of confidence that a build is worth further testing, and the number of bad builds delivered to QA can be minimized. Based on a set of Python APIs, you can write scripts that install and run applications on emulators and devices, send specific keystrokes, and take screenshots. When combined with the JUnit unit testing framework, you can develop powerful automated test suites.

### **Testing on the Emulator Versus the Device**

When you can get your hands on the actual device your users have, focus your testing there. However, devices and the service contracts that generally come with them can be expensive. Your test team cannot be expected to set up test environments on every carrier or every country where your users use your application. There are times when the Android emulator can reduce costs and improve testing coverage. Some of the benefits of using the emulator include

- Ability to simulate devices when they are not available or in short supply
- Ability to test difficult test scenarios not feasible on live devices
- Ability to be automated like any other desktop software

### **Testing Before Devices Are Available Using the Emulator**

Developers often target up-and-coming devices or platform versions not yet available to the general public. These devices are often highly anticipated, and developers who are ready with applications for these devices on Day 1 of release often experience a sales bump because fewer applications are available to these users—less competition, more sales.

The latest version of the Android SDK is usually released to developers several months prior to when the general public receives over-the-air updates. Also, developers can sometimes gain access to preproduction devices through carrier and manufacturer

developer programs. However, developers and testers should be aware of the dangers of testing on preproduction devices: The hardware is generally beta quality. The final technical specifications and firmware can change without notice. Release dates can slip, and the device might never reach production.

When preproduction devices cannot be acquired, testers can do some functional testing using emulator AVD configurations that attempt to closely match the target platform, thus lessening the risks for a compact testing cycle when these devices go live and allowing developers to release applications faster.

### **Understanding the Dangers of Relying on the Emulator**

Unfortunately, the emulator is more of a generic Android device that pretends at many of the device internals—despite all the options available within the AVD configuration.



#### **Tip**

Consider developing a document describing the specific AVD configurations used for testing different device configurations as part of the test plan.

The emulator does not represent the specific implementation of the Android platform that is unique to a given device. It does not use the same hardware to determine signal, networking, or location information. The emulator can pretend to make and receive calls and messages, or take pictures or video. At the end of the day, it doesn't matter if the application works on the emulator if it doesn't work on the actual device.

### **Testing Strategies: Black- and White-Box Testing**

The Android tools provide ample tools for black box and white box testing:

- Black-box testers might require only testing devices and test documentation. For black-box testing, it is even more important that the testers have a working knowledge of the specific devices, so providing device manuals and technical specifications also aids in more thorough testing. In addition to such details, knowing device nuances as well as device standards can greatly help with usability testing. For example, if a dock is available for the device, knowing that it's either landscape or portrait mode is useful.
- White-box testing has never been easier on mobile. White-box testers can leverage the many affordable tools, including the Eclipse development environment (which is free) and the many debugging tools available as part of the Android SDK. White-box testers use the Android emulator, DDMS, and ADB especially. They can also take advantage of the powerful unit-testing framework and the Hierarchy Viewer for user interface debugging. For these tasks, the tester requires a computer with a development environment similar to the developer's as well as knowledge of Java, Python, and the various typical tools available for developers.

## Testing Mobile Application Servers and Services

Although testers often focus on the client portion of the application, they sometimes neglect to thoroughly test the server portion. Many mobile applications rely on networking or the cloud. If your application depends on a server or remote service to operate, testing the server side of your application is vital. Even if the service is not your own, you need to test thoroughly against it so you know it behaves as the application expects it to behave.



### Warning

Users expect applications to be available any time, day or night, 24/7. Minimize server or service downtimes and make sure the application notifies the users appropriately (and doesn't crash and burn) if a service is unavailable. If the service is outside your control, it might be worthwhile to look at what Service Level Agreements are offered.

Here are some guidelines for testing remote servers or services:

- Version your server builds. You should manage server rollouts like any other part of the build process. The server should be versioned and rolled out in a reproducible way.
- Use test servers. Often, QA tests against a mock server in a controlled environment. This is especially true if the live server is already operational with real users.
- Verify scalability. Test the server or service under load, including stress testing (many users, simulated clients).
- Test the server security (hacking, SQL injection, and such).
- Ensure data transmissions to and from the server are secure and not easily sniffed (SSL, HTTPS, valid certificates).
- Ensure that your application handles remote server maintenance or service interruptions gracefully—scheduled or otherwise.
- Test your old clients against new servers to ensure expected, graceful application behavior. Consider versioning your server communications and protocols in addition to your client builds.
- Test server upgrades and rollbacks and develop a plan for how you are going to inform users if and when services are down.

These types of testing offer yet another opportunity for automated testing to be employed.

## Testing Application Visual Appeal and Usability

Testing a mobile application is not only about finding dysfunctional features, but also about evaluating the usability of the application. Report areas of the application that lack visual appeal or are difficult to navigate or use. We like to use the walking-and-chewing-gum analogy when it comes to mobile user interfaces. Mobile users frequently do not

give the application their full attention. Instead, they walk or do something else while they use it. Applications should be as easy for the user as chewing gum.

### Tip

Consider conducting usability studies to collect feedback from people who are not familiar with the application. Relying solely on the product team members, who see the application regularly, can blind the team to application flaws.

## Leveraging Third-Party Standards for Android Testing

Make a habit to try to adapt traditional software-testing principles to mobile. Encourage quality assurance personnel to develop and share these practices within your company.

Again, no certification programs are specifically designed for Android applications at this time; however, nothing is stopping the mobile marketplaces from developing them. Consider looking over the certification programs available in other mobile platforms, such as the extensive testing scripts and acceptance guidelines used by Windows, Apple, and BREW platforms and adjusting them for your Android applications. Regardless of whether you plan to apply for a specific certification, making an attempt to conform to well-recognized quality guidelines can improve your application's quality.

## Handling Specialized Test Scenarios

In addition to functional testing, there are a few other specialized testing scenarios that any QA team should consider.

## Testing Application Integration Points

It's necessary to test how the application behaves with other parts of the Android operating system. For example:

- Ensuring that interruptions from the operating system are handled properly (incoming messages, calls, and powering off)
- Validating Content Provider data exposed by your application, including such uses as through a Live Folder
- Validating functionality triggered in other applications via an Intent
- Validating any known functionality triggered in your application via an Intent
- Validating any secondary entry points to your application as defined in `AndroidManifest.xml`, such as application shortcuts
- Validating alternative forms of your application, such as App Widgets
- Validating service-related features, if applicable

## Testing Application Upgrades

When possible, perform upgrade tests of both the client and the server or service side of things. If upgrade support is planned, have development create a mock upgraded

Android application so that QA can validate that data migration occurs properly, even if the upgraded application does nothing with the data.



### Tip

Users receive Android platform updates over the air on a regular basis. The platform version on which your application is installed might change over time. Some developers have found that firmware upgrades have broken their applications, necessitating upgrades.

Always retest your applications when a new version of the SDK is released, so that you can upgrade users before your applications have a chance to break in the field.

If your application is backed by an underlying database, you'll want to test versioning your database. Does a database upgrade migrate existing data or delete it? Does the migration work from all versions of the application to the current version, or just the last version?

### Testing Device Upgrades

Applications are increasingly using the cloud and backup services available on the Android platform. This means that users who upgrade their devices can seamlessly move their data from one device to another. So if they drop their smartphone in a hot tub or crack their tablet screen, their application data can often be salvaged. If your application leverages these services, make sure you test whether these transitions work.

### Testing Product Internationalization

It's a good idea to test internationalization support early in the development process—both the client and the server or services. You're likely to run into some problems in this area related to screen real estate and issues with strings, dates, times, and formatting.



### Tip

If your application will be localized for multiple languages, test in a foreign language—especially a verbose one. The application might look flawless in English but be unusable in German, where words are generally longer.

### Testing for Conformance

Make sure to review any policies, agreements, and terms to which your application must conform and make sure your application complies. For example, Android applications must by default conform to the Android Developer Agreement and, when applicable, the Google Maps terms of service. Other distribution means and add-on packages may add further terms that your application must abide by.

### Installation Testing

Generally speaking, installation of Android applications is straightforward; however, you need to test installations on devices with low resources and low memory as well as test

installation from the specific marketplaces when your application “goes live.” If the manifest install location allows external media, be sure to test various low or missing resource scenarios.

## **Backup Testing**

Don’t forget to test features that are not readily apparent to the user, such as the backup and restore services and the sync features discussed in *Android Wireless Application Development Volume II: Advanced Topics*.

## **Performance Testing**

Application performance matters in the mobile world. The Android SDK has support for calculating performance benchmarks within an application and monitoring memory and resource usage. Testers should familiarize themselves with these utilities and use them often to help identify performance bottlenecks and dangerous memory leaks and misused resources.

One common performance issue we see frequently with new Android developers is trying to do everything on the main UI thread. Time- and resource-intensive work, such as network downloads, XML parsing, graphics rendering, and other such tasks, should be moved “off” the main UI thread so that the user interface remains responsive. This helps avoid so-called “Force Close” (or FC) issues and negative reviews saying as much.

The `Debug` class (`android.os.Debug`) has been around since Android was first released. This class provides a number of methods for generating trace logs that can then be analyzed using the `traceview` test tool. Android 2.3 introduced a new class called `StrictMode` (`android.os.StrictMode`) that can be used to monitor your applications, track down latency issues, and banish ANRs from your apps. There’s also a great write-up on the Android Developer’s blog, available at <http://goo.gl/1d5Ka>.

Here’s another good example of a common performance issue we see from new Android application developers. Many do not realize that, by default, Android screens (backed by activities) are restarted every time a screen orientation change happens. Unless the developer takes the appropriate actions, nothing is cached by default. Even basic applications really need to take care of how their lifecycle management works. Tools are available to do this efficiently. Yet, we frequently run into very inefficient ways of doing this—usually due to not handling lifecycle events at all.

## **Testing Application Billing**

Billing is too important to leave to guesswork. Test it. You’ll notice a lot of test applications on the Android Market. Remember to specify that your application is a test app.

## **Testing for the Unexpected**

Regardless of the workflow you design, understand that users do random, unexpected things—on purpose and by accident. Some users are “button mashers,” whereas others forget to set the keypad lock before putting the device in their pocket, resulting in a weird set of key presses. Rotating the screen frequently, sliding a physical keyboard in

and out, or fiddling with other settings often triggers unexpected configuration changes. A phone call or text message inevitably comes in during the farthest, most-remote edge cases. Your application must be robust enough to handle this. The Exerciser Monkey command-line tool can help you test for this type of event.

### Testing to Increase Your Chances of Being a “Killer App”

Every mobile developer wants to develop a “killer app”—those applications that go viral, rocket to the top of the charts, and make millions a month. Most people think that if they just find the right idea, they’ll have a killer app on their hands. Developers are always scouring the top-ten lists and the Android Market’s Editor’s Choice category, trying to figure out how to develop the next great app. But let us tell you a little secret: If there’s one thing that all “killer apps” share, it’s a higher-than-average quality standard. No clunky, slow, obnoxious, or difficult-to-use application ever makes it to the big leagues. Testing and enforcing quality standards can mean the difference between a mediocre application and a killer app.

If you spend any time examining the mobile marketplace, you’ll notice a number of larger mobile development companies publish a variety of high-quality applications with a shared look and feel. These companies leverage user interface consistency as well as shared and above-average quality standards to build brand loyalty and increase market share, while hedging their bets that perhaps just one of their many applications will have that magical combination of great idea and quality design. Other, smaller companies often have the great ideas but struggle with the quality aspects of mobile software development. The inevitable result is that the mobile marketplace is full of fantastic application ideas badly executed with poor user interfaces and crippling defects.

## Leveraging Android Tools for Android Application Testing

The Android SDK and developer community provide a number of useful tools and resources for application testing and quality assurance. You might want to leverage the following tools during this phase of your development project:

- The physical devices for testing and bug reproduction
- The Android emulator for automated testing and testing of builds when devices are not available
- The Android DDMS tool for debugging and interaction with the emulator or device, as well as for taking screenshots
- The ADB tool for logging, debugging, and shell access tools
- The Exerciser Monkey command-line tool for stress testing of input (available via the ADB shell)
- The logcat command-line tool, which can be used to view log data generated by the application (best used with debug versions of your application)

- The `traceview` application, which can be used to view and interpret the tracing log files you can generate from your app
- The `sqlite3` command-line tool for application database access (available via ADB shell)
- The Hierarchy Viewer for user interface debugging, performance tweaking, and for pixel-perfect screenshots of the device
- The `layoutopt` tool, which can be used to optimize the layout resources of an application
- The `bmgr` command-line tool, which can help test backup management features of your application, if applicable

It should be noted that although we have used the Android tools, such as the Android emulator and DDMS debugging tools with Eclipse, these are standalone tools that can be used by quality assurance personnel without the need for source code or a development environment.



### Tip

The tools discussed in Chapters 4, “Mastering the Android Development Tools,” and in the appendixes of this book are valuable not just to developers; these tools provide testers with much more control over device configuration.

## Avoiding Silly Mistakes in Android Application Testing

Here are some of the frustrating and silly mistakes and pitfalls that Android testers should try to avoid:

- Not testing the server or service components used by an application as thoroughly as the client side.
- Not testing with the appropriate version of the Android SDK (device versus development build versions).
- Not testing on the device and assuming the emulator is enough.
- Not testing the live application using the same system that users use (billing, installation, and such). Buy your own app.
- Neglecting to test all entry points to the application.
- Neglecting to test in different coverage areas and network speeds.
- Neglecting to test using battery power. Don’t always have the device plugged in.

## Summary

In this chapter, we armed you—the keepers of application quality—with real-world knowledge for testing Android applications. Whether you’re a team of one or one hundred, testing your applications is critical for project success. Luckily, the Android SDK provides a number of tools for testing applications, as well as a powerful unit-testing framework. By following standard quality assurance techniques and leveraging these tools, you can ensure that the application you deliver to your users is the best it can be.

## References and More Information

Android Dev Guide: Testing and Instrumentation:

[http://developer.android.com/guide/topics/testing/testing\\_android.html](http://developer.android.com/guide/topics/testing/testing_android.html)

Android Dev Guide: Testing:

<http://developer.android.com/guide/developing/testing/index.html>

Android Tools: UI/Application Exerciser Monkey:

<http://developer.android.com/guide/developing/tools/monkey.html>

The monkeyrunner Tool:

[http://developer.android.com/guide/developing/tools/monkeyrunner\\_concepts.html](http://developer.android.com/guide/developing/tools/monkeyrunner_concepts.html)

Wikipedia on Software Testing:

[http://en.wikipedia.org/wiki/Software\\_testing](http://en.wikipedia.org/wiki/Software_testing)

Software Testing Help:

<http://www.softwaretestinghelp.com>

# Publishing Your Android Application

After you've developed and tested your application, the next logical step is to publish it so that other people can enjoy it. You might even want to make some money. A variety of distribution opportunities are available to Android application developers. Many developers choose to sell their applications through mobile marketplaces such as Google's Android Market. Others develop their own distribution mechanisms—for example, they might sell their applications from a website. Regardless, developers should consider which distribution options they plan to use during the application design and development process, because some distribution choices might require code changes or impose restrictions on content.

## Choosing the Right Distribution Model

The application distribution methods you choose to employ depend on your goals and target users. Here are some questions you should ask yourself:

- Is your application ready for prime time or are you considering a beta period to iron out the kinks?
- Are you trying to reach the broadest audience, or have you developed a vertical market application? Determine who your users are, which devices they are using, and their preferred methods for seeking out and downloading applications.
- How will you price your application? Is it freeware or shareware? Are the payment models (single payment versus subscription model versus ad-driven revenue) you require available on the distribution mechanisms you want to leverage?
- Where do you plan to distribute? Verify that any application markets you plan to use are capable of distributing within those countries or regions.
- Are you willing to share a portion of your profits? Distribution mechanisms such as the Android Market take a percentage of each sale in exchange for hosting your application for distribution and collecting application revenue on your behalf.

- Do you require complete control over the distribution process or are you willing to work within the boundaries and requirements imposed by third-party marketplaces? This might require compliance with further license agreements and terms.
- If you plan to distribute yourself, how will you do so? You might need to develop more services to manage users, deploy applications, and collect payments. If so, how will you protect user data? What trade laws must you comply with?
- Have you considered creating a free trial version of your application? If the distribution system under consideration has a return policy, consider the ramifications. You need to ensure that your application has safeguards to minimize the number of users that buy your app, use it, and return it for a full refund. For example, a game might include safeguards such as a free trial version and a full-scale version with more game levels than could possibly be completed within the refundable time period.

## Protecting Your Intellectual Property

You've spent time, money, and effort to build a valuable Android application. Now you want to distribute it but perhaps you are concerned about reverse engineering of trade secrets and software piracy. As technology rapidly advances, it's impossible to perfectly protect against either.

If you're accustomed to developing Java applications, you might be familiar with code-obfuscation tools. These are designed to strip easy-to-read information from compiled Java byte codes, making the decompiled application more difficult to understand and reverse engineer. Some tools, such as ProGuard (<http://proguard.sourceforge.net>), support Android applications because they can run after the .jar file is created and before it's converted to the final package file used with Android. ProGuard support is built in to Android projects created with the Android tools.

The Android Market also supports a licensing service called the License Verification Library (LVL). This is available as a Google API add-on, but works on Android versions 1.5 and higher. It only applies to paid applications distributed through Android Market. It requires application support—code additions—to be fully utilized, and you should seriously consider obfuscating your code. The service's primary purpose is to verify that a paid application installed on a device was properly purchased by the user. You can find out more at the Android Developer website: <http://goo.gl/Tbvjh>.

Finally, the Android Market supports a form of copy protection via a check box when you publish your application. The method used isn't well documented currently. However, you can also use your own copy protection methods or those available through other markets if this is a huge concern for you or your company.



### Tip

You'll learn lots of strategies for protecting your applications from software piracy in *Android Wireless Application Development Volume II: Advanced Topics*.

## Billing the User

Unlike some other mobile platforms you might have used, the Android SDK does not currently provide built-in billing APIs that work directly from within applications or charge the users' wireless bill directly. Instead, billing APIs are normally add-on APIs that are provided by the distribution channels. For example, Android Market uses Google checkout for processing application payments.

If an application needs to charge ad-hoc fees for goods sold within the application (that is, ringtones, music, e-books, and more), the application developer must implement an in-app billing mechanism. Android Market provides an add-on API for implementing in-app billing support within any applications published to the Android Market (<http://goo.gl/emzJo>).



### Tip

You'll learn how to use Android Market's in-app billing APIs in the second volume of this book.

Rolling your own billing system? Most Android devices can leverage the Internet, so using online billing services and APIs—PayPal, Google Checkout, and Amazon, to name a few—is a common choice. Check with your preferred billing service to make sure it specifically allows mobile use and that the billing methods your application requires are available, feasible, and legal for your target users. Similarly, make sure any distribution channels you plan to use allow these billing mechanisms (as opposed to their own).

## Leveraging Ad Revenue

Another method to make money from users is to have an ad-supported mobile business model. Android itself has no specific rules against using advertisements within applications. However, different markets may impose their own rules on what's allowed. For instance, Google's own AdMob service allows developers to place ads within their applications. (Read more at <http://goo.gl/q77mK>.) Several other companies provide similar services.

## Collecting Statistics Regarding Your Application

Before you publish, you may want to consider adding some statistics collection to your application to determine how your users use it. You could write your own statistics-collection mechanisms, or you can use third-party add-ons such as Google Analytics for Android. Ensure that you always inform your users if you are collecting information about them, and incorporate your plans into your clearly defined end user license agreement (EULA) and privacy policy. Statistics can help you not just see how many people are using your application, but how they are actually using it.



### Tip

You'll learn how to use Google Analytics for Android in the second volume of this book.

Now let's look at the steps you need to take to package and publish your application.

## Packaging Your Application for Publication

Developers must take several steps when preparing an Android application for publication and distribution. Your application must also meet several important requirements imposed by the marketplaces. The following steps are required for publishing an application:

1. Prepare and perform a release candidate build of the application.
2. Verify that all requirements for the marketplace are met, such as configuring the Android manifest file properly. For example, make sure the application name and version information are correct and the `debuggable` attribute is set to `false`.
3. Package and digitally sign the application.
4. Test the packaged application release thoroughly.
5. Publish the application.

The preceding steps are required but not sufficient to guarantee a successful deployment. Developers should also take these steps:

1. Thoroughly test the application on all target handsets.
2. Turn off debugging, including `Log` statements and any other logging.
3. Verify permissions, making sure to add ones for services used and remove any that aren't used, regardless of whether they are enforced by the handsets.
4. Test the final, signed version with all debugging and logging turned off.

Now, let's explore each of these steps in more detail, in the order they might be performed.

### Preparing Your Code for Packaging

An application that has undergone a thorough testing cycle might need changes made to it before it is ready for a production release. These changes convert it from a debuggable, preproduction application into a release-ready application.

#### Setting the Application Name and Icon

An Android application has default settings for the icon and label. The icon appears in the application launcher and can appear in various other locations, including marketplaces. As such, an application is required to have an icon. You should supply alternate icon `drawable` resources for various screen resolutions. The label, or application name, is also displayed in similar locations and defaults to the package name. You should choose a short, user-friendly name that displays under the application icon in launcher screens.

#### Versioning the Application

Next, proper versioning is required, especially if updates could occur in the future. The version name is up to the developer. The version code, though, is used internally by the

Android system to determine if an application is an update. You should increment the version code for each new update of an application. The exact value doesn't matter, but it must be greater than the previous version code. Versioning within the Android manifest file is discussed in Chapter 6, "Defining Your Application Using the Android Manifest File."

### Verifying the Target Platforms

Make sure your application sets the `<uses-sdk>` tag in the Android manifest file correctly. This tag is used to specify the minimum and target platform versions that the application can run on. This is perhaps the most important setting after the application name and version information.

### Configuring the Android Manifest for Market Filtering

If you plan to publish through the Android Market, you should read up on how this distribution system uses certain tags within the Android manifest file to filter applications available to users. Many of these tags, such as `<supports-screens>`, `<uses-configuration>`, `<uses-feature>`, `<uses-library>`, `<uses-permission>`, and `<uses-sdk>`, were discussed in Chapter 6. Set each of these items carefully, because you don't want to accidentally put too many restrictions on your application. Make sure you test your application thoroughly after configuring these Android manifest file settings. For more information on how Android Market filters work, see <http://goo.gl/kbI3o>.

### Preparing Your Application Package for the Android Market

The Android Market has strict requirements on application packages. When you upload your application to the Android Market website, the package is verified and any problems are communicated to you. Most often, problems occur when you have not properly configured your Android manifest file.

The Android Market uses the `android:versionName` attribute of the `<manifest>` tag within the Android manifest file to display version information to users. It also uses the `android:versionCode` attribute internally to handle application upgrades. The `android:icon` and `android:label` attributes must also be present because both are used by the Android Market to display the application name to the user with a visual icon.

#### Warning

The Android SDK allows the `android:versionName` attribute to reference a string resource. The Android Market, however, does not. An error is generated if a string resource is used.

### Disabling Debugging and Logging

Next, you should turn off debugging and logging. Disabling debugging involves removing the `android:debuggable` attribute from the `<application>` tag of the

`AndroidManifest.xml` file or setting it to `false`. You can turn off the logging code within Java in a variety of different ways, from just commenting it out to using a build system that can do this automatically.



### Tip

A common method for conditionally compiling debug code is to use a class interface with a single, public, static, final Boolean that's set to `true` or `false`. When this is used with an `if` statement and set to `false`, because it's immutable, the compiler should not include the unreachable code, and it certainly won't be executed. We recommend using some method other than just commenting out the Log lines and other debug code.



### Tip

If you don't specify the `android:debuggable` attribute, incremental builds will automatically turn it on and export/release builds will leave it off. Specifying the value as `true` will also cause export/release builds to actually do a debug build.

## Verifying Application Permissions

Finally, the permissions used by the application should be reviewed. Include all permissions that the application requires, and remove any that are not used. Users appreciate this.

## Packing and Signing Your Application

Now that the application is ready for publication, the file package—the `.apk` file—needs to be prepared for release. The package manager of an Android device will not install a package that has not been digitally signed. Throughout the development process, the Android tools have accomplished this through signing with a debug key. The debug key cannot be used for publishing an application to the wider world. Instead, you need to use a true key to digitally sign the application. You can use the private key to digitally sign the release package files of your Android application, as well as any upgrades. This ensures that the application (as a complete entity) is coming from you, the developer, and not some other source (imposters!).



### Warning

A private key identifies the developer and is critical to building trust relationships between developers and users. It is very important to secure private key information.

The Android Market requires that your application's digital signature validity period end after October 22, 2033. This date might seem like a long way off and, for mobile, it certainly is. However, because an application must use the same key for upgrading and because applications that want to work closely together with special privileges and trust

relationships must also be signed with the same key, the key could be chained forward through many applications. Thus, Google is mandating that the key be valid for the foreseeable future so application updates and upgrades are performed smoothly for users.



### Note

Finding a third-party certificate authority that will issue a key valid for such a long duration can be a challenge, so self-signing is the most straightforward signing solution. Within the Android Market, there is no benefit to using a third-party certificate authority.

Although self-signing is typical of Android applications, and a certificate authority is not required, creating a suitable key and securing it properly is critical. The digital signature for Android applications can impact certain functionality. The expiry of the signature is verified at installation time, but after it's installed, an application continues to function even if the signature has expired.

You can export and sign your Android package file from within Eclipse using the Android Development plug-in as follows (or you can use the command-line tools):

1. In Eclipse, right-click the appropriate application project and choose the Android Tools, Export Signed Android Application... option. (Alternatively, you can choose Export, expand the Android section, and choose Export Android Application, which defaults to a signed application.)
2. Click the Next button.
3. Select the project to export (the one you right-clicked before is the default).
4. On the keystore selection screen, choose the Create New Keystore option and enter a file location (where you want to store the key) as well as a password for managing the keystore. (If you already have a keystore, choose Browse to pick your keystore file and then enter the correct password.)



### Warning

Make sure you choose strong passwords for the keystore. Remember where the keystore is located, too. The same one is required to publish an upgrade to your application. If it's checked in to a revision-control system, the password helps protect it. However, you should consider adding an extra layer of privilege required to get to it.

5. Click the Next button.
6. On the Key Creation screen, enter the details of the key, as shown in Figure 19.1.
7. Click the Next button.
8. On the Destination and Key/Certificate Checks screen, enter a destination for the application package file.
9. Click the Finish button.

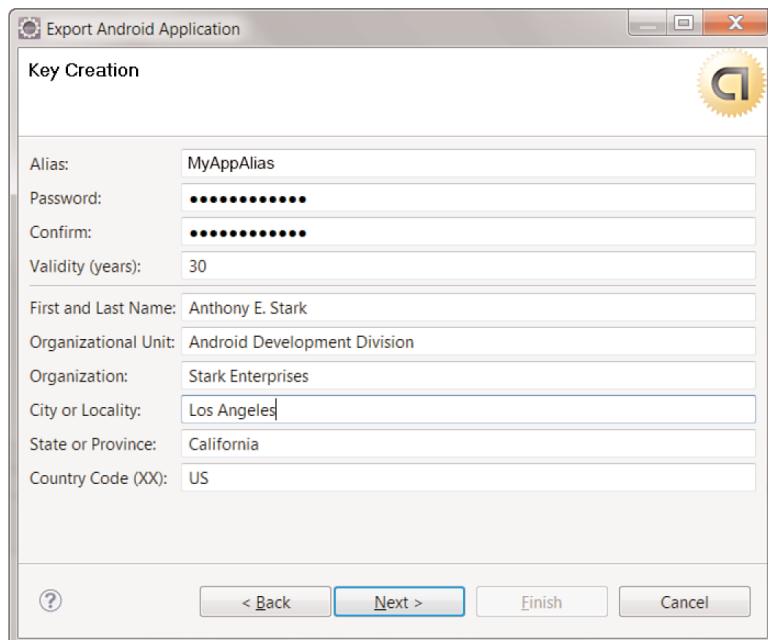


Figure 19.1 Creating a new key for exporting a signed Android application in Eclipse.

You have now created a fully signed and certified application package file. The application package is ready for publication. For more information about signing, see the Android Developer website: <http://goo.gl/iROjP>.



#### Note

If you are not using Eclipse and the Android Development plug-in, you can use the `keytool` and `jarsigner` command-line tools available within the JDK, in addition to the `zipalign` utility provided with the Android SDK, to create a suitable key and sign an application package file (`.apk`). Although `zipalign` is not directly related to signing, it optimizes the application package for more efficient use on Android. The ADT plug-in for Eclipse runs `zipalign` automatically after the signing step.

## Testing the Release Version of Your Application Package

Now that you have configured your application for production, you should perform a full final testing cycle, paying special attention to subtle changes to the installation process. An important part of this process is to verify that you have disabled all debugging features and that logging has no negative impact on the functionality and performance of the application.

## Distributing Your Application

Now that you've prepared your application for publication, it's time to get it out to users—for fun and profit. Before you publish, you may want to consider setting up an application website, tech support email address, help and feedback forum, Twitter/Facebook/Google+/social network *du jour* account, and any other infrastructure you may want or need to support your published application.

## Publishing on the Android Market

The Android Market is the most popular mechanism for distributing Android applications at the time of this writing. This is where your typical user purchases and downloads applications. As of this writing, it's available to most, but not all, Android devices. As such, we show you how to check your package for preparedness, sign up for a developer account, and submit your application for sale on the Android Market.

### Note

The Android Market is updated frequently. We have made every attempt to provide the latest steps for uploading and managing applications. However, these steps and the user interfaces described in this section may change at any time. Please review the Android Market website (<https://market.android.com/publish/>) for the latest information.

## Signing Up for a Developer Account on the Android Market

To publish applications through the Android Market, you must register as a developer. This accomplishes two things. It verifies who you are to Google and signs you up for a Google Checkout account, which is used for the billing of Android applications.

### Note

As of this writing, only developers ("Merchants") residing in certain approved countries may sell priced applications on the Android Market due to international laws, as described here: <http://goo.gl/JfUw0>. Developers from many other countries can register for Publisher accounts, but they may only publish free applications at this time. For a complete list of supported publisher countries, see <http://goo.gl/sGHHZ>.

To sign up for an Android Market developer account, you need to follow these steps:

1. Go to the Android Market sign-up website at <http://market.android.com/publish/signup>, as shown in Figure 19.2.
2. Sign in with the Google Account you want to use. (At this time, you cannot change the associated Google Account, but you can change the contact email addresses for applications independently.)

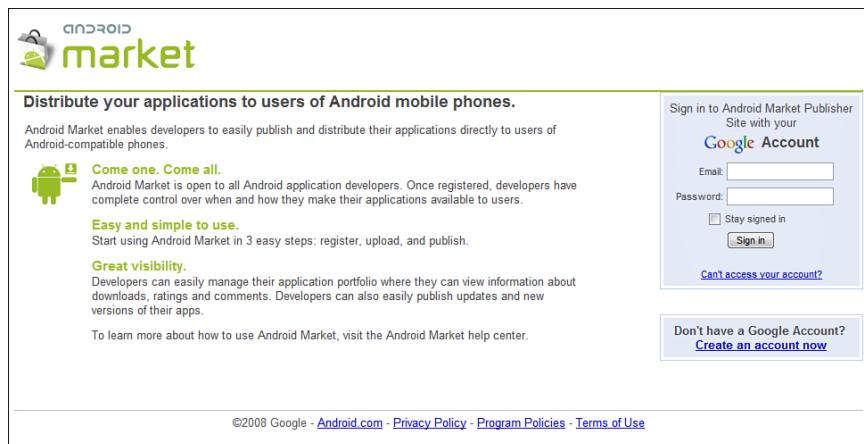


Figure 19.2 The Android Market publisher sign-up page.

3. Enter your developer information, including your name, email address, and website, as shown in Figure 19.3.

The screenshot shows the 'Getting Started' section of the Android Market developer listing page. It includes a list of requirements: 'Create a developer profile', 'Pay a registration fee (\$25.00) with your credit card (using Google Checkout)', and 'Agree to the [Android Market Developer Distribution Agreement](#)'. Below this is a 'Listing Details' section where developers can enter their profile information. Fields include 'Developer Name' (with a note: 'Will appear to users under the name of your application'), 'Email Address', 'Website URL', 'Phone Number' (with a note: 'Include country code and area code. [why do we ask for this?](#)'), and 'Email updates' (with a note: 'Contact me occasionally about development and Market opportunities'). At the bottom is a blue 'Continue »' button and a copyright notice: '© 2010 Google - [Android Market Developer Distribution Agreement](#) - [Google Terms of Service](#) - [Privacy Policy](#)'.

Figure 19.3 The Android Market developer listing page.

4. Confirm your registration payment (as of this writing, \$25 USD). Note that Google Checkout is used for registration payment processing.
5. Signing up and paying to be an Android Developer also creates a mandatory Google Checkout Merchant account for which you also need to provide information. This account is used for payment processing purposes.
6. Agree to link your credit card and account registration to the Android Market Developer Distribution Agreement. The basic agreement (U.S. version) is available for review at <http://goo.gl/V0BXD>. Always print out the actual agreement you sign as part of the registration process, in case it changes in the future.

When you successfully complete these steps, you are presented with the home screen of the Android Market, which also confirms that the Google Checkout Merchant account was created.

## Uploading Your Application to the Android Market

Now that you have an account registered for publishing applications through Android Market and a signed application package, you are ready to upload it for publication.

From the main page of the Android Market website, sign in and click the Upload Application button, shown in Figure 19.4.

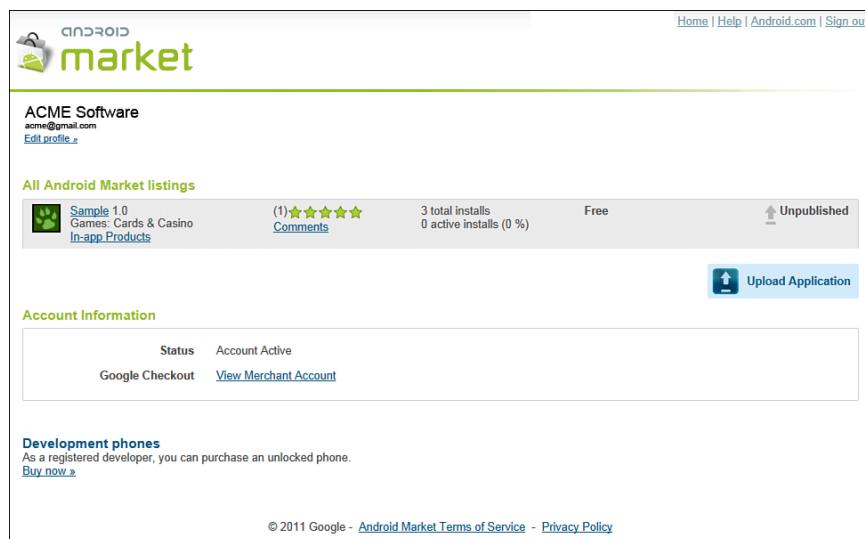


Figure 19.4 Android Market developer application listings.

From this page, you can configure developer account settings, see your payment transaction history, and manage your published applications. In order to publish a new application, click the Upload Application button on this page. A form is presented for uploading the application package file(s), or APKs, associated with your product (see Figure 19.5).

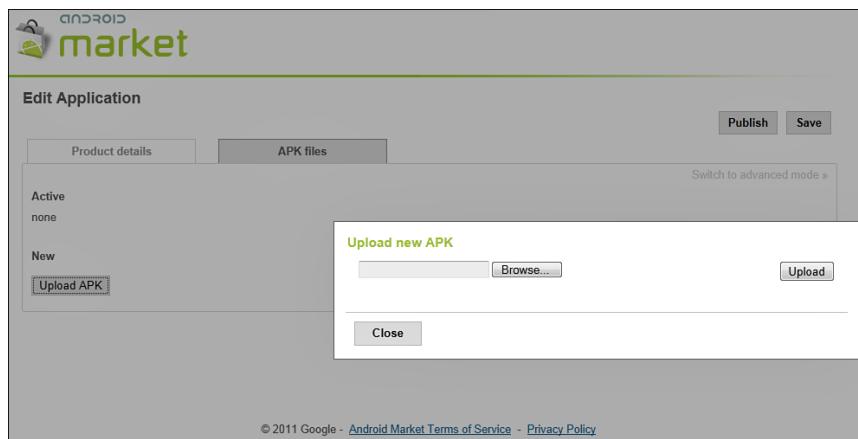


Figure 19.5 Android Market application upload form.

Back on the Product details tab, you can fill in the important fields associated with your application.

## Uploading Application Marketing Assets

The Product Details tab associated with your application begins with the Upload Assets section (see Figure 19.6). Here, you can perform the following tasks:

- Provide a high-resolution version of your application icon.
- Upload screenshots and promotional graphics for the market listing.
- Link to an optional promotional video of the application from YouTube.
- Define your marketing settings, including the ability to opt out of Android Market and Google marketing opportunities.

## Configuring Application Listing Details

The Product Details tab associated with your application continues with the Listing Details section (see Figure 19.7). Here, you can do the following:

- Specify the listing languages you want to support.
- Specify the application title, description, recent changes, and promotional text in those specific languages.
- Specify the application's type and category.

**Edit Application**

**Product details**    **APK files**    **Publish**    **Save**

**Upload assets**

Screenshots at least 2    Add a screenshot:    

Screenshots:  
320 x 480, 480 x 800,  
480 x 854, 1280 x 800  
24 bit PNG or JPEG (no alpha)  
Full bleed, no border in art  
You may upload screenshots in  
landscape orientation. The thumbnails will  
appear to be rotated, but the actual  
images and their orientations will be  
preserved.

High Resolution Application Icon [\[Learn More\]](#)    Add a hi-res application icon:    

High Resolution Application Icon:  
512w x 512h  
32 bit PNG or JPEG  
Maximum: 1024 KB

Promotional Graphic optional    Add a promotional graphic:    

Promo Graphic:  
180w x 120h  
24 bit PNG or JPEG (no alpha)  
No border in art

Feature Graphic optional    Add a feature graphic:    

Feature Graphic:  
1024w x 500h  
24 bit PNG or JPEG (no alpha)  
Will be downszied to mini or micro

Promotional Video optional    Add a promotional video link:

Promotional Video:  
Enter YouTube URL.

Marketing Opt-Out  Do not promote my application except in Android Market and in any Google-owned online or mobile properties. I understand that any changes to this preference may take sixty days to take effect.

Figure 19.6 Android Market application upload form (Upload Assets).

**Listing details**

Language  [\[Learn More\]](#) | Star sign (\*) indicates the default language.

Title (English)  0 characters (30 max)

Description (English)  0 characters (4000 max)

Recent Changes (English) [\[Learn More\]](#)  0 characters (500 max)

Promo Text (English)  0 characters (80 max)

Application Type

Category

Figure 19.7 Android Market application upload form (Listing Details).



### Tip

Spend the time to set the application type and category fields appropriately so that your application reaches its intended audience. Incorrectly categorized applications do not sell well. For a complete list of types and categories, see the Android Market Help listing at <http://goo.gl/SngkX>.

## Configuring Application Publishing Options

The Product Details tab associated with your application continues with the Publishing Options section (see Figure 19.8). Here, you can perform the following tasks:

- Specify your application's copy protection settings.
- Specify the content rating (maturity level) of your application.
- Set the pricing type (free or paid) of the application (permanent setting, cannot be changed later).
- Set the default price of your application in U.S. Dollars, if applicable.
- Set the default price of your application in various other currencies, if applicable, using a one-time currency conversion (you can adjust these values as desired).



### Note

Currently a 30% transaction fee is imposed for hosting applications within the Android Market. Prices can range from \$0.99 to \$200 (USD), and similar ranges are available in other supported currencies. For more details, see <http://goo.gl/ZoDMK>.

- Include or exclude specific devices, as necessary (above and beyond market filters).

## Configuring Application Contact and Consent Information

The last portions of the Product Details tab are the Contact Information and Consent sections (see Figure 19.9). Here, you can do the following:

- Specify your developer website, email, and phone number.



### Note

You can change these contact settings on an app-by-app basis, which allows for great support flexibility when your company is publishing multiple applications.

- Legally certify that your application meets the Android Content Guidelines, as defined at <http://goo.gl/416A0>. (Required.)
- Legally acknowledge that your application is subject to U.S. export laws and that you agree to comply with the laws and export rules that the Android Market uses to publish your application internationally. (Required.)

**Publishing options**

**Copy Protection**

- Off (Application can be copied from the device)
- On (Helps prevent copying of this application from the device. Increases the amount of memory on the phone required to install the application.)

The copy protection feature will be deprecated soon, please use [licensing service](#) instead.

**Content Rating** [\[Learn More\]](#)

- High Maturity
- Medium Maturity
- Low Maturity
- Everyone

**Pricing**

- Free  Paid

Setting the price to Free is permanent; you cannot change to a price later. [\[Learn More\]](#)

Set a price for each country/region

Default price

Automatically populate the fields with a one-time conversion to local currencies from the default price with the current exchange rate

All Countries

<input checked="" type="checkbox"/> Argentina	<input checked="" type="checkbox"/> Latvia
<input checked="" type="checkbox"/> Australia	<input checked="" type="checkbox"/> Lithuania
<input checked="" type="checkbox"/> Austria	<input checked="" type="checkbox"/> Luxembourg
<input checked="" type="checkbox"/> Belgium	<input checked="" type="checkbox"/> Malta

<Parts of this form deleted for brevity>

<input checked="" type="checkbox"/> Ireland	<input checked="" type="checkbox"/> Switzerland
<input checked="" type="checkbox"/> Israel	<input checked="" type="checkbox"/> Taiwan
<input checked="" type="checkbox"/> Italy	<input checked="" type="checkbox"/> Thailand
<input checked="" type="checkbox"/> Japan	<input checked="" type="checkbox"/> Ukraine
<input checked="" type="checkbox"/> Kenya	<input checked="" type="checkbox"/> United Kingdom
	<input checked="" type="checkbox"/> United States

<Parts of this form deleted for brevity>

**Supported Devices** [\[Learn More\]](#)

This application is only available to devices with these features, as defined in your application manifest.

This application is available to over 0 devices.

Figure 19.8 Android Market application upload form (Publishing Options).

**Contact information**

Website

Email

Phone

**Consent**

This application meets [Android Content Guidelines](#)

I acknowledge that my software application may be subject to United States export laws, regardless of my location or nationality. I agree that I have complied with all such laws, including any requirements for software with encryption functions. I hereby certify that my application is authorized for export from the United States under these laws. [\[Learn More\]](#)

Figure 19.9 Android Market application upload form (Contact Information and Consent).

## Publishing Your Application on the Android Market

Finally, you are ready to click the Publish button. Your application appears in the Android Market almost immediately. After publication, you can see statistics including ratings, reviews, downloads, and active installs in the Your Android Market Listings section of the main page on your developer account. You can also access crash reports and other information once your application is downloaded by users.



### Tip

Receiving crash reports from a specific device? Check your application's market filters in the Android manifest file. Are you including and excluding the appropriate devices? You can also exclude specific devices by adjusting the Supported Devices settings of the application listing.

## Managing Your Application on the Android Market

Once you've published your application on the Android Market, you will need to manage it. Some considerations include understanding how the Android Market return policy works, managing application upgrades, and, if necessary, removing your application from publication.

### Understanding the Android Market Application Return Policy

The Android Market currently has a 15-minute refund policy on applications. That is to say, a user can use an application for 15 minutes and then return it for a full refund. However, this only applies to the first download and first return. If a particular user has already returned your application and wants to "try it again," he or she must make a final purchase—and can't return it a second time. Although this limits abuse, you should still be aware that if your application has limited reuse appeal, you might find that you have a return rate that's too high and need to pursue other methods of monetization.



### Tip

As a developer, you can also issue refunds to specific users using the Google Checkout Merchant Center. Find out more here: <http://goo.gl/J1oei>.

### Upgrading Your Application on the Android Market

You can upgrade existing applications on the Market from the developer account page. Simply upload a new version of the same application using the Android manifest file tag, `android:versionCode`. When you publish it, users receive an Update Available notification, prompting them to download the upgrade.

 **Warning**

Application updates must be signed with the same private key as the original application. For security reasons, the Android package manager does not install the update over the existing application if the key is different. This means you need to keep the key corresponding with the application in a secure, easy-to-find location for future use.

**Removing Your Application from the Android Market**

You can also use the unpublish action to remove the application from the Market from the developer account. The unpublish action is immediate, but the application entry on the Market application might be cached on handsets that have viewed or downloaded the application. Keep in mind that unpublishing the application makes it unavailable to new users but does not remove it from existing users' devices.

## Publishing Using Other Alternatives

The Android Market is not the only place available to distribute your Android applications. Many alternative distribution mechanisms are available to developers. Application requirements, royalty rates, and licensing agreements vary by store. Third-party application stores are free to enforce whatever rules they want on the applications they accept, so read the fine print carefully. They might enforce content guidelines, require additional technical support, and enforce digital signing requirements. Only you and your team can determine which are suitable for your specific needs.

 **Tip**

Android is an open platform, which means there is nothing preventing a handset manufacturer or an operator (or even you) from developing their own Android application store.

Here are a few alternative marketplaces where you might consider distributing your Android applications:

- **Amazon Appstore** is an example of an Android-specific distribution website for free and paid applications (<http://amazon.com/appstore>).
- **Soc.io Mall** (formerly AndAppStore) is an Android-specific distribution for free applications, e-books, and music using an on-device store (<http://mall.soc.io>).
- **Handango** distributes mobile applications across a wide range of devices with various billing models (<http://www.handango.com>).
- **SHOP4APPS** is an example of an application storefront run by the device manufacturer Motorola for its customers (<http://goo.gl/qUfSJ>).
- **V CAST Apps** is an example of a curated store run by the carrier Verizon for its mobile subscribers (<http://goo.gl/g8is8>).

**Tip**

New market opportunities arise all the time. Check out our article on the top ten places to sell your Android apps at <http://goo.gl/YNaHr>. Wikipedia keeps a helpful list of digital distribution platforms for mobile devices, which includes app stores for Android and other mobile platforms (<http://goo.gl/gMpyG>).

## Self-Publishing Your Application

You can distribute Android applications directly from a website or server. This method is most appropriate for vertical market applications, content companies developing mobile marketplaces, and big-brand websites wanting to drive users to their branded Android applications. It can also be a good way to get beta feedback from end users.

Although self-distribution is perhaps the easiest method of application distribution, it might also be the hardest to market, protect, and make money. The only requirement for self-distribution is to have a place to host the application package file.

The downside of self-distribution is that end users must configure their devices to allow packages from unknown sources. This setting is found under the Applications section of the device's Settings application, as shown in Figure 19.10. This option is not available on all consumer devices in the market.

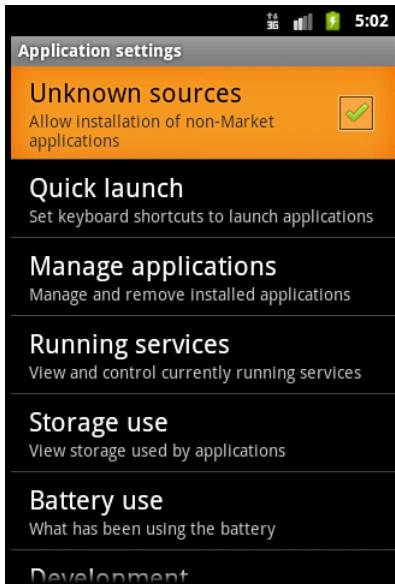


Figure 19.10 Settings application showing the required check box for downloading from unknown sources.

After that, the final step the user must make is to enter the URL of the application package into the web browser on the handset and download the file (or click a link to it). When the file is downloaded, the standard Android install process occurs, asking the user to confirm the permissions and, optionally, confirm an update or replacement of an existing application if a version is already installed.

## Summary

You've now learned how to design, develop, test, and deploy professional-grade Android applications. In this final chapter, you learned how to prepare your application package for publication using a variety of revenue models. Whether you publish through the Android Market, alternative markets, your own website, or some combination of these options, you can now build a robust application from the ground up and distribute it for profit (or fame!).

So, now it's time to go out there, fire up Eclipse, and build some amazing applications. We want to encourage you to think outside of the box. The Android platform leaves the developer with a lot more freedom and flexibility than most mobile platforms. Take advantage of this. Use what works and reinvent what doesn't. You might just find yourself with a killer app.

Finally, if you're so inclined, we'd love to know about all the exciting applications you're building. You'll find our contact information in the Introduction at the beginning of this book. Best of luck!

## References and More Information

The Android Market website:

<http://market.android.com/>

Android Dev Guide: "Market Filters":

<http://developer.android.com/guide/appendix/market-filters.html>

*This page intentionally left blank*

# VI

## Appendices

- A** The Android Emulator Quick-Start Guide
- B** The Android DDMS Quick-Start Guide
- C** Eclipse IDE Tips and Tricks

*This page intentionally left blank*

# A

# The Android Emulator Quick-Start Guide

The most useful tool provided with the Android Software Development Kit (SDK) is the emulator. Developers use the emulator to quickly develop Android applications for a variety of hardware. This Quick-Start Guide is not a complete documentation of the emulator commands. Instead, it is designed to get you up and running with common tasks. Please see the emulator documentation provided with the Android SDK for a complete list of features and commands.

The Android emulator is integrated with Eclipse using the Android Development Tools Plug-in for the Eclipse integrated development environment (IDE). The emulator is also available within the `/tools` directory of the Android SDK and you can launch it as a separate process. The best way to launch the emulator is by using the Android Virtual Device Manager.

## Simulating Reality: The Emulator's Purpose

The Android emulator (shown in Figure A.1) simulates a real device environment where your applications run. As a developer, you can configure the emulator to closely resemble the devices on which you plan to deploy your applications.

Here are some tips for using the emulator effectively from the start:

- You can use keyboard commands to easily interact with the emulator.
- Mouse clicking within the emulator window works, as does scrolling and dragging. So do the keyboard arrow buttons. Don't forget the side buttons, such as the volume control. These work, too.
- If your computer has an Internet connection, so does your emulator. The browser works. You can toggle networking using the F8 key.
- Different Android platform versions show slightly different underlying user experiences (the basics of the Android operating system) on the emulator. For example, older platform targets have a basic Home screen and use an application drawer to

store installed applications, whereas the newer smartphone-centric platform versions such as Android 4.0+ use sleeker controls and an improved Home screen, and the Honeycomb (Android 3.0+) platform targets include the new holographic theme and action bar navigation. The emulator uses the basic user interface, which is frequently overridden, or skinned, by manufacturers and carriers. In other words, the operating system features of the emulator might not match what real users see.

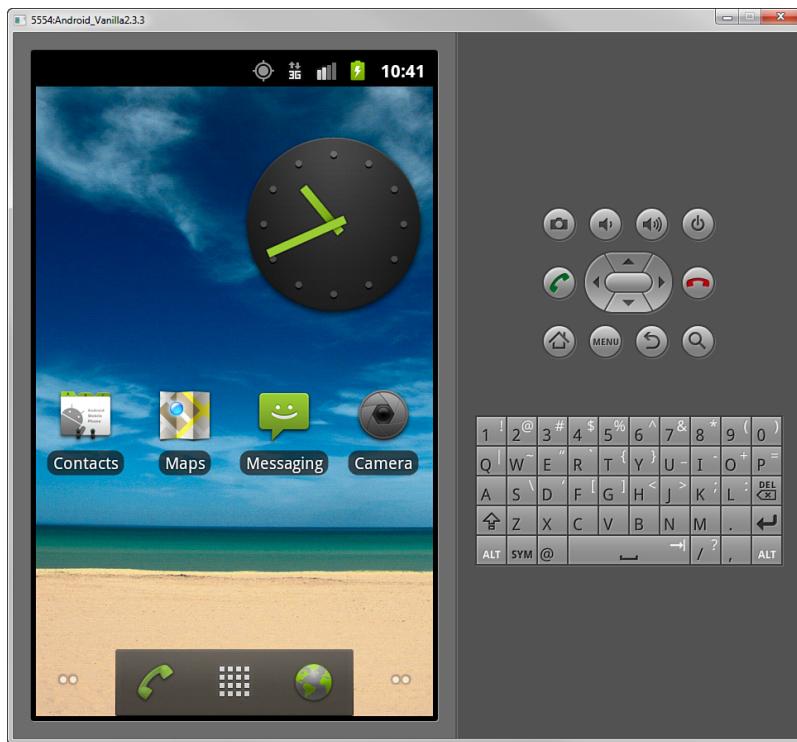


Figure A.1 A typical Android emulator.

- The Settings application can be useful for managing system settings. You can use the Settings application to configure the user settings available within the emulator, including networking, screen options, and locale options.
- The Dev Tools application can be useful for setting development options. These include many useful tools, from a terminal emulator to a list of installed packages. Additionally, tools for accounts and sync testing are available. JUnit tests can be launched directly from here.

- To switch between portrait and landscape modes of the emulator, use the 7 and 9 keys on the numeric keypad (or the Ctrl+F11 and Ctrl+F12 keys).
- You can use the F6 key to emulate a trackball with your mouse. This takes over exclusive control of your mouse, so you must use F6 to get control back again.
- The Menu button is a context menu for the given screen. Keep in mind that newer devices do not always have the physical keys such as Home, Menu, Back, and Search.
- With regard to application lifecycle: To easily stop an application, just press Home (on the emulator) and you'll get `onPause()` and `onStop()` Activity lifecycle events. To resume, launch the application again. To pause the application, press the Power button (on the emulator). Only the `onPause()` method will be called. You'll need to press the Power button a second time to activate the display to be able to unlock the emulator to see the `onResume()` method call.
- Notifications such as incoming SMS messages appear in the notification bar, along with indicators for simulated battery life, signal strength and speed, and so on.

### Warning

One of the most important things to remember when working with the emulator is that it is a powerful tool, but it is no substitute for testing on the true target device. The emulator often provides a much more consistent user experience than a physical device, which moves around in the physical world, through tunnels and cell signal dead zones, with many other applications running and sucking down battery and resources. Always budget time and resources to thoroughly exercise your applications on target physical devices and in common situations as part of your testing process.

## Working with Android Virtual Devices (AVDs)

The Android emulator is not a real device, but a generic Android system simulator for testing purposes. Developers can simulate different types of Android devices by creating Android Virtual Device (AVD) configurations.

### Tip

It can be helpful to think of an AVD as providing the emulator's personality. Without an AVD, an emulator is an empty shell, not unlike a CPU with no attached peripherals.

Using AVD configurations, Android emulators can simulate

- Different target platform versions
- Different screen sizes and resolutions
- Different input methods

- Different network types, speeds, and strengths
- Different underlying hardware configurations
- Different external storage configurations

Each emulator configuration is unique, as described within its AVD profile, and stores its data persistently, including installed applications, modified settings, and the contents of its emulated SD card. A number of emulator instances with different AVD configurations is shown in Figure A.2.



Figure A.2 AVD configurations described in different emulator settings.

## Using the Android Virtual Device Manager

To run an application in the Android emulator, you must configure an Android Virtual Device (AVD). To create and manage AVDs, you can use the Android Virtual Device Manager from within Eclipse (available as part of the ADT plug-in) or use the `android` command-line tool provided with the Android SDK in the `/tools` subdirectory. Each

AVD configuration contains important information describing a specific type of Android device, including the following:

- The friendly, descriptive name for the configuration
- The target Android platform version
- The screen size, aspect ratio, and resolution
- Hardware configuration details and features, including how much RAM is available, which input methods exist, and optional hardware details such as cameras and location sensor support
- Simulated external storage (virtual SD cards)

Figure A.3 illustrates how you can use the Android Virtual Device Manager to create and manage AVD configurations.

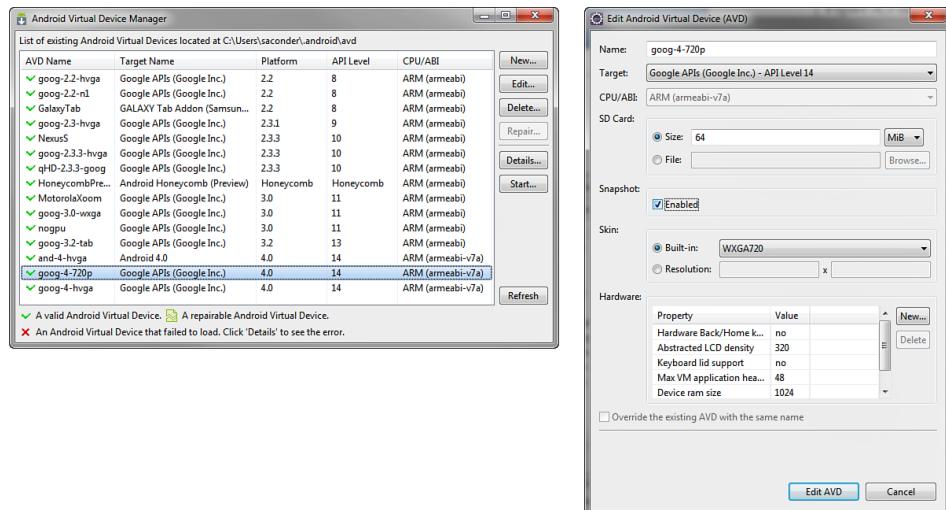


Figure A.3 The Android Virtual Device Manager (left) can be used to create AVD configurations (right).

## Creating an AVD

Follow these steps to create an AVD configuration within Eclipse:

1. Launch the Android Virtual Device Manager from within Eclipse by clicking the little green Android icon with the arrow () on the toolbar. You can also launch it by selecting Window, Android Virtual Device Manager from the Eclipse menu.
2. Click the Virtual Devices menu item on the left menu (Figure A.3, left). The configured AVDs are displayed as a list.
3. Click the New button to create a new AVD (Figure A.3, right).

4. Choose a name for the AVD. If you are trying to simulate a specific device, you might want to name it as such. For example, a name such as “NexusOne2.2\_Style” might refer to an AVD that simulates the Nexus One handset running the Android 2.2 platform with the Google APIs.
5. Choose a build target. This represents the version of the Android platform running on the emulator. The platform is represented by the API level. For example, to support Android 2.2, use API Level 8. However, this is also where you choose whether or not to include the optional Google APIs. If your application relies on the Maps application and other Google Android services, you should choose the target with the Google APIs. For a complete list of API levels and which Android platforms they represent, see <http://d.android.com/guide/appendix/api-levels.html>.
6. Choose an SD card capacity. This capacity can be configured in kibibytes or mibibytes. Each SD card image takes up space on your hard drive and takes a long time to generate; don’t make your card capacities too large, or they will hog your hard drive space. Choose a reasonable size, such as a 1024MiB or less. The minimum is 9MiB. Just make sure you have adequate disk space on your development computer and choose an appropriate size for your testing needs. If you’re dealing with images or videos, you may need to allocate much more capacity.
7. Choose a skin. This determines the screen characteristics to emulate. For each target platform, there are a number of predefined skins (WXGA, HVGA, and so on) that represent common Android device characteristics to make this easy. Different skins are available for different build targets. You can also set your own screen settings if none of the predefined skins match your requirements.
8. Configure or modify any hardware characteristics that do not match the defaults. Sometimes the predefined skins automatically set some of these characteristics for you, such as screen density. You may also want to change the input methods available for a given AVD.
9. Click the Create AVD button and wait for the operation to complete. Because the Android Virtual Device Manager formats the memory allocated for SD card images, creating an AVD configuration sometimes takes a few moments.
10. Click Finish.

## Exploring AVD Display Options

Different Android devices have different screen characteristics. Testing your application in emulators configured to simulate appropriate screen sizes and resolutions is crucial.

The first display option is the AVD’s skin, which maps to the device resolution in pixels. Different target platform versions support different skins. For example, Android 1.6 (API Level 4) supports the following skins: QVGA, HVGA (default), WVGA800, and WVGA854. Android 3.0 (API Level 13) supports a default tablet skin WXGA. For those

not familiar with these sorts of screen type definitions, they are standardized graphics display resolutions. You can find their pixel dimensions defined at <http://goo.gl/Vd7mq>. For example, WXGA is defined as 1280×768 pixels. Sometimes a display resolution will vary slightly for “wide screens” and such. For example, WVGA is technically 480×800 (when in portrait mode, natural for a phone), but because there is also a widescreen version of this display, we use the more specific definitions of WVGA800 (480×800) and FWVGA854 (480×854).

The next display option you’ll want to consider is the screen density. Whereas the skin maps to the number of pixels that make up the screen, the screen density represents how many pixels per inch (dpi) are displayed by the device screen. For the Android platform and emulator purposes, screens are placed into density categories. The baseline density for emulators is a medium density of 160 dpi. A high-density display is defined as 240 dpi, and extra-high-density is 320 dpi. By default, your emulator will emulate a medium-density device, but you can modify this by changing the hardware property Abstracted LCD Density (more on this in a moment). You can use this property to emulate low, medium, high, and ultra-high screen densities.

### Tip

You can also adjust the density of an existing AVD with an emulator window using emulator startup line options such as `-scale` and `-dpi-device`. For more information, see the Emulator Startup Options documentation at the Android Developer website: <http://d.android.com/guide/developing/devices/emulator.html#startup-options>. You can supply these startup options as part of your project’s Debug configuration, or use them on the command line.

Want your emulator to more closely resemble a specific device? Check the manufacturer’s developer website: They may have custom skins you can install and use with the Android emulator. Some geeks even make their own skins to customize their development environment.

### Creating AVDs with Custom Hardware Settings

As mentioned earlier, you can specify specific hardware configuration settings within your AVD configurations. You need to know what the default settings are to know whether you need to override them. Some of the hardware options available are shown in Table A.1.

Table A.1 Important Hardware Profile Options

Hardware Property Option	Description	Default Value
<code>Device RAM Size</code> <code>hw.ramSize</code>	Physical RAM on the device, in megabytes.	Depends on target and options, from 96 to 1024
<code>Touch-screen Support</code> <code>hw.touchScreen</code>	Touch screen exists on the device.	Yes

Table A.1 **Continued**

<b>Hardware Property Option</b>	<b>Description</b>	<b>Default Value</b>
Trackball Support hw.trackBall	Trackball exists on the device.	Yes
Keyboard Support hw.keyboard	QWERTY keyboard exists on the device.	Yes
GPU Emulation hw.gpu.enabled	Emulate OpenGL ES GPU.	Yes
D-Pad Support hw.dPad	Directional pad exists on the device.	Yes
GSM Modem Support hw.gsmModem	GSM modem exists in the device.	Yes
Camera Support hw.camera	Camera exists on the device.	No
Camera Pixels (Horizontal) hw.camera. maxHorizontalPixels	Maximum horizontal camera pixels.	640
Camera Pixels (Vertical) hw.camera. maxVerticalPixels	Maximum vertical camera pixels.	480
Number of Emulated Web Cameras hw.webcam.count	Number of web cameras.	6
GPS Support hw.gps	GPS exists on the device.	Yes
Battery Support hw.battery	Device can run on a battery.	Yes
Accelerometer Support hw.accelerometer	Accelerometer exists on the device.	Yes
Audio Recording Support hw.audioInput	Device can record audio.	Yes
Audio Playback Support hw.audioOutput	Device can play audio.	Yes
SD Card Support hw.sdCard	Device supports removable SD cards.	Yes
Cache Partition Support disk.cachePartition	Device supports cache partition.	Yes

Hardware Property Option	Description	Default Value
Cache Partition Size disk.cachePartition.size	Device cache partition size in megabytes.	66
Abstracted LCD Density hw.lcd.density	Generalized screen density.	Depends on target and chosen screen resolution: 120, 160, 240, 320
Max VM App Heap Size vm.heapSize	Maximum heap size an application can allocate before being killed by the operating system.	Depends on target and options: 16, 24, 48

 **Tip**

You can save time, money, and a lot of grief by spending a bit of time upfront configuring AVDs that closely match the hardware upon which your application will run. Share the specific settings with your fellow developers and testers. We often create device-specific AVDs and name them after the device.

## Launching the Emulator with a Specific AVD

After you have configured the AVD you want to use, you are ready to launch the emulator. Although there are a number of ways to do this, there are four main ways you will likely use on a regular basis:

- From within Eclipse, you can configure the application's Debug or Run configuration to use a specific AVD.
- From within Eclipse, you can configure the application's Debug or Run configuration to enable the developer to choose an AVD manually upon launch.
- From within Eclipse, you can launch an emulator directly from within the Android Virtual Device Manager.
- The emulator is available within the `/tools` directory of the Android SDK and can be launched as a separate process from the command line (generally only necessary if you are not using Eclipse).

## Maintaining Emulator Performance

Like most emulators, the Android emulator is notoriously slow. That said, there are a few ways you can help ensure the best and speediest emulator experience possible. Here are a few tips along those lines:

- Enable the snapshot feature in your AVDs. Then, before you start using your AVD, launch it once, let it boot up, and shut it down to set a baseline snapshot. This is especially important with the newest platform versions such as Honeycomb. Subsequent launches will be faster and more stable. You can even turn off saving a new snapshot to speed up exiting and it will continue to use the old snapshot.
- Launch your emulator instances before you need them, such as when you first launch Eclipse, so that when you're ready to debug, they're already running.
- Keep the emulator running in the background between debugging sessions in order to quickly install, reinstall, and debug your applications. This saves valuable minutes of waiting for the emulator to boot up. Instead, simply launch the Debug configuration from Eclipse and the debugger reattaches.
- Keep in mind that application performance is much slower when the debugger is attached. This applies to both running in the emulator and on a device.
- If you've been using an emulator for testing many apps, or just need a very clean environment, consider re-creating the AVD from scratch. This will give you a new environment clean of any past changes or modifications. This can help speed up the emulator, too, if you have lots of apps installed.

## Configuring Emulator Startup Options

The Android emulator has a number of configuration options above and beyond those set in the AVD profile. These options are configured in the Eclipse Debug and Run configurations for your specific applications, or when the emulator is launched from the command line. Some emulator startup settings include network speed and latency, media settings, the ability to disable boot animation upon startup, and numerous other system settings. There are also debugging settings, such as support for proxy servers, DNS addresses, and other details. For a complete list of emulator startup options, consult the Android emulator documentation: <http://d.android.com/guide/developing/tools/emulator.html#startup-options>

## Launching an Emulator to Run an Application

The most common way you launch the emulator involves launching a specific emulator instance with a specific AVD configuration, either through the Android Virtual Device Manager or by choosing a Run or Debug configuration for your project in Eclipse and installing or reinstalling the latest incarnation of your application.



### Tip

Remember that you can create Run configurations and Debug configurations separately, with different options, using different startup options and even different AVDs.

To create a Debug configuration for a specific project within Eclipse, take the following steps:

1. Choose Run, Debug Configurations (or right-click the project and Choose Debug As...).
2. Double-click on Android Application.
3. Name your Debug configuration (we often use the project name).
4. Choose the project by clicking on the Browse button.
5. Switch to the Target tab and choose the appropriate Deployment Target Selection Mode. Either choose a specific AVD to use with the emulator (only those matching your application's target SDK are shown) or select the Manual option to be prompted upon launch to choose an AVD on the fly.



### Tip

If you have Android devices connected via USB when you attempt to run or debug your application from Eclipse, you will be prompted to choose your target at runtime despite having selected a specific AVD. This enables you to redirect the install or reinstall operation to a device other than an emulator. You can always force this behavior by choosing the Manual option for your Deployment Target Selection Mode on the Target tab of your Debug configuration. We find setting the mode to Manual helps if you switch between debugging on emulators and devices frequently, and setting a specific target is more useful when only debugging with a specific AVD emulator instance.

6. Configure any emulator startup options on the Target tab. You can enter any options not specifically shown on the tab as normal command-line options in the Additional Emulator Command Line Options field.

The resulting Debug configuration might look something like Figure A.4.

You can create Run configurations in a very similar fashion. If you set a specific AVD for use in the Deployment Target Selection Mode settings, that AVD is used with the emulator whenever you debug your application in Eclipse. However, if you chose the Manual option, you are prompted to select an AVD from the Android Device Chooser when you first try to debug the application, as shown in Figure A.5. After you have launched that emulator, Eclipse pairs it with your project for the duration of your debugging session.

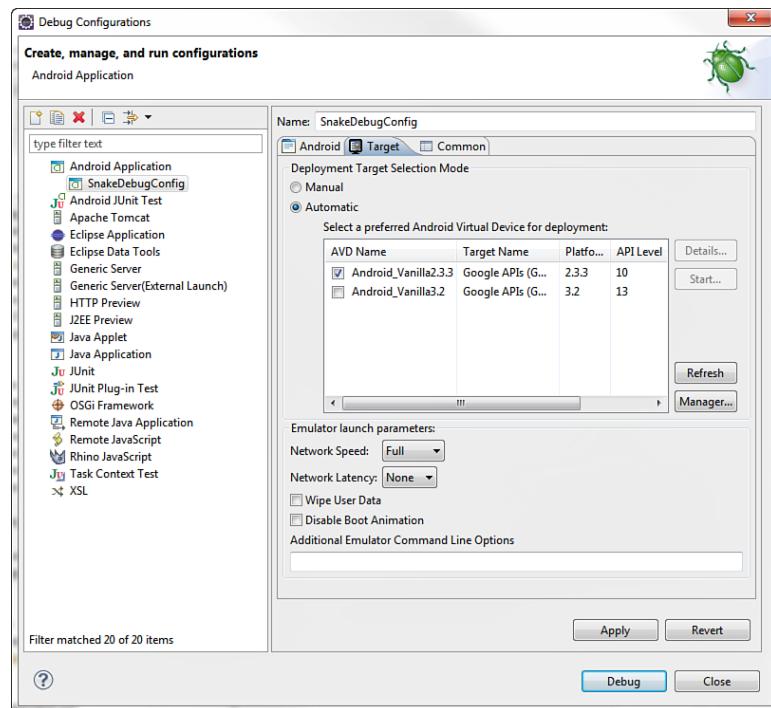


Figure A.4 Creating a Debug configuration in Eclipse.

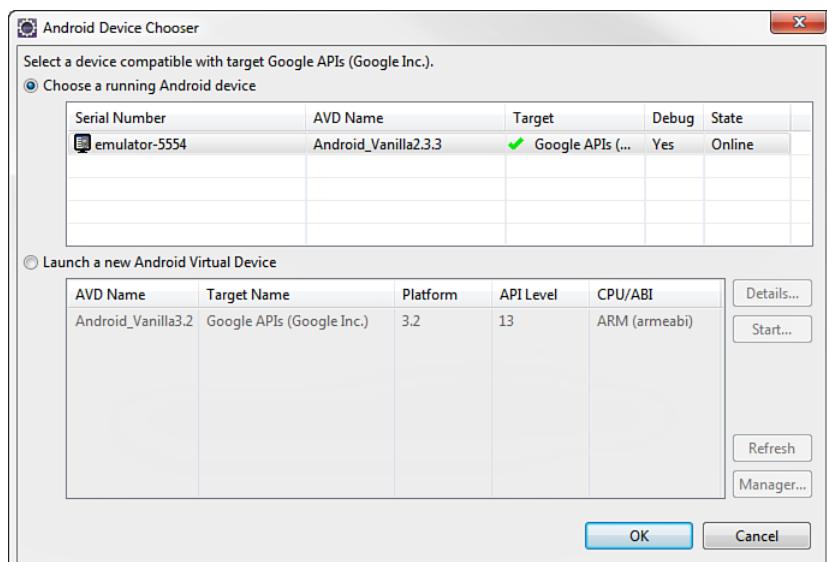


Figure A.5 The Android Device Chooser.

## Launching an Emulator from the Android Virtual Device Manager

Sometimes you just want to launch an emulator on the fly—for example, to have a second emulator running to interact with your first emulator to simulate calls, text messages, and such. In this case, you can simply launch it from the Android Virtual Device Manager. To do this, take the following steps:

1. Launch the Android Virtual Device Manager from within Eclipse ( on the toolbar. You can also launch it by selecting Window, Android Virtual Device Manager from the Eclipse menu.
2. Click the Virtual Devices menu item on the left menu. The configured AVDs are displayed as a list.
3. Select an existing AVD configuration from the list or create a new AVD that matches your requirements.
4. Hit the Start button.
5. Configure any launch options necessary.
6. Hit the Launch button. The emulator now launches with the AVD you requested.

### Warning

You cannot run multiple instances of the same AVD configuration simultaneously. If you think about it, this makes sense because the AVD configuration keeps the state and persistent data.

## Configuring the GPS Location of the Emulator

To develop and test applications that use Google Maps support with location-based services, you need to begin by creating an AVD with a target that includes the Google APIs. After you have created the appropriate AVD and launched the emulator, you need to configure its location. The emulator does not have location sensors, so the first thing you need to do is seed your emulator with GPS coordinates.

To seed your emulator with pretend coordinates, launch your emulator (if it is not already running) with an AVD supporting the Google Maps add-ins and follow these steps:

**In the Emulator:**

1. Press the Home key to return to the Home screen.
2. Find and launch the Maps application.
3. Click through the various startup dialogs, if this is the first time you've launched the Maps application.
4. Choose the My Location menu item (  ).

**In Eclipse:**

5. Click the DDMS perspective in the top-right corner of Eclipse.
6. You see an Emulator Control pane on the left side of the screen. Scroll down to the Location Control.
7. Manually enter the longitude and latitude of your location. Note that they are in reverse order. For example, Yosemite Valley has the coordinates Longitude: -119.588542 and Latitude: 37.746761.
8. Click Send.

Back in the emulator, notice that the map now shows the location you seeded. Your screen should now display your location as Yosemite, as shown in Figure A.6. This location persists across emulator launches.

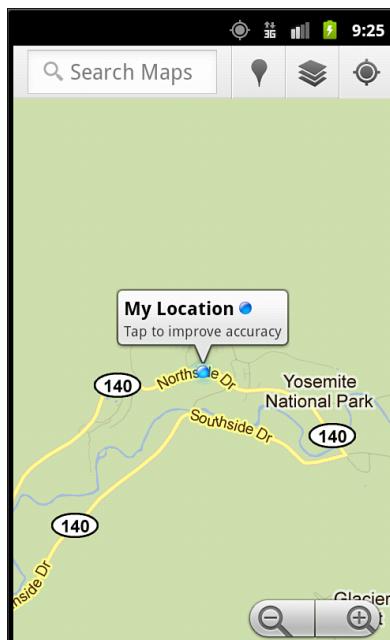


Figure A.6 Setting the location of the emulator to Yosemite Valley.

You can also use GPX 1.1 coordinate files to send a series of GPS locations through DDMS to the emulator, if you prefer.



### Tip

Wondering where we got the coordinates for Yosemite? To find a specific set of coordinates for use, you can go to <http://maps.google.com>. Navigate to the location you want the coordinates for. Next, right-click the location and choose, “What’s here?” or “Drop LatLng Marker.” If you choose the first method, the latitude and longitude will be placed on the search field. If you choose the second method, a marker with the latitude and longitude displayed in it will be placed on the map itself.

## Calling Between Two Emulator Instances

You can have two emulator instances call each other using the Dialer application provided on the emulator. The emulator’s “phone number” is its port number, which can be found in the title bar of the emulator window. To simulate a phone call between two emulators, you must perform the following steps:

1. Launch two different AVDs so two emulators are running simultaneously. (Using the Android AVD and SDK Manager is easiest.)
2. Note the port number of the emulator you want to receive the call.
3. In the emulator that makes the call, launch the Dialer application.
4. Type the port number you noted as the number to call. Press Enter (or Send).
5. You see (and hear) an incoming call on the receiving emulator instance. Figure A.7 shows an emulator with port 5556 (left) using the Dialer application to call the emulator on port 5554 (right).
6. Answer the call by pressing Send or swiping across the Dialer app.
7. Pretend to chat for a bit. Figure A.8 shows a call in progress.
8. You can end either emulator call at any time by pressing the End key.

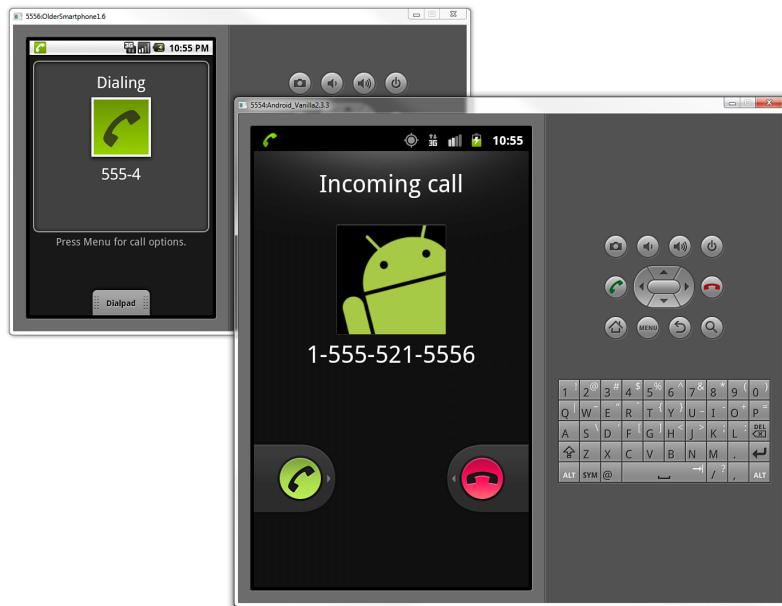


Figure A.7 Simulating a phone call between two emulators.

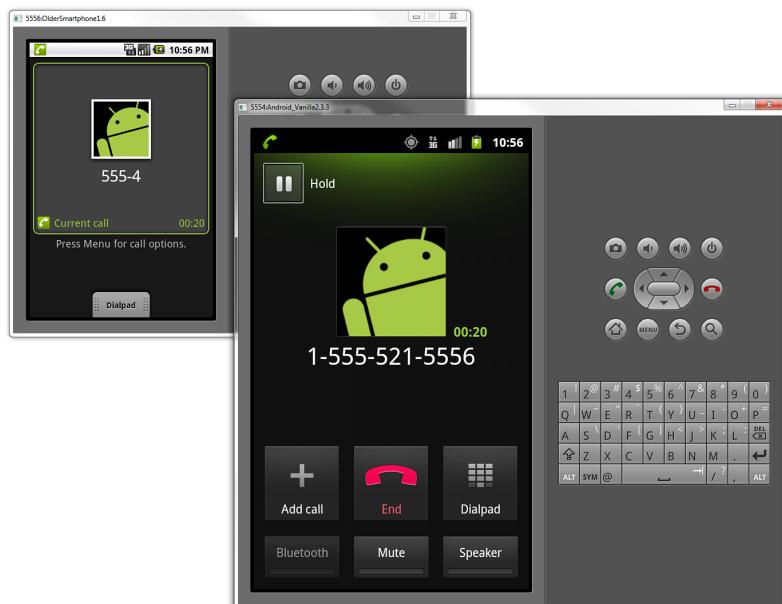


Figure A.8 Two emulators with a phone call in progress.

## Messaging between Two Emulator Instances

You can send SMS messages between two emulators, exactly as previously described for simulating calls, by using the emulator port numbers as SMS addresses. To simulate a text message between two emulators, you must perform the following steps:

1. Launch two instances of the emulator.
2. Note the port number of the emulator you want to receive the text message.
3. In the emulator that sends the text, launch the Messaging application.
4. Type the port number you noted as the “To” field for the text message. Enter a text message, as shown in Figure A.9 (left). Press the Send button.
5. You see (and hear) an incoming text message on the receiving emulator instance. Figure A.9 (right, top) shows an emulator with port 5554 receiving a text message from the emulator on port 5556.
6. View the text message by pulling down the notification bar or launching the Messaging app.
7. Pretend to chat for a bit. Figure A.9 (right, bottom) shows a text message conversation in progress.

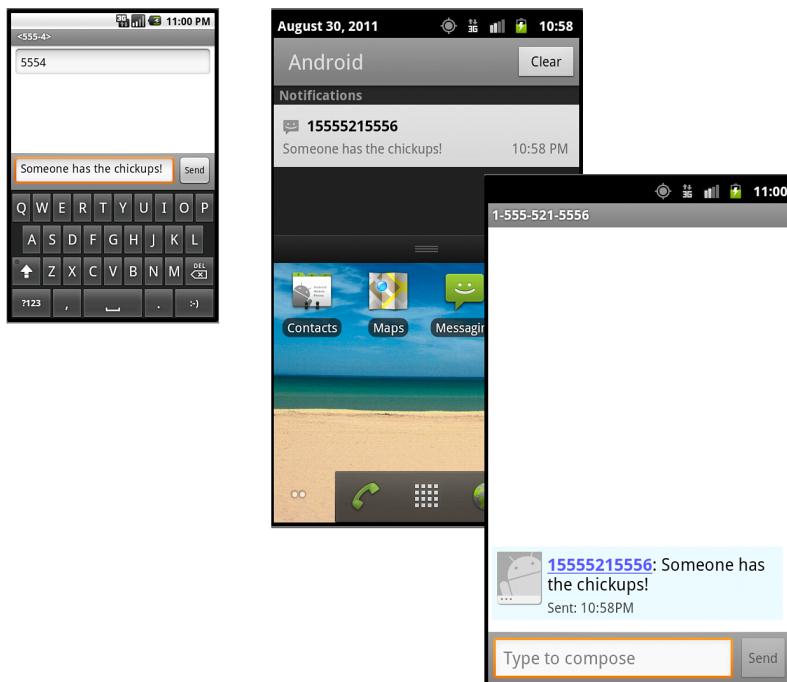


Figure A.9 Emulator at port 5556 crafting a text message to send to another emulator at port 5554.

## Interacting with the Emulator through the Console

In addition to using the DDMS tool to interact with the emulator, you can also connect directly to the emulator console using a Telnet connection and then issue commands. For example, to connect to the Emulator console of the emulator using port 5554, you would do the following:

```
telnet localhost 5554
```

You can use the Emulator console to issue commands to the emulator. To end the session, just type `quit` or `exit`. You can shut down this instance of the emulator using the `kill` command.

### Using the Console to Simulate Incoming Calls

You can simulate incoming calls to the emulator from specified numbers. The console command for issuing an incoming call is

```
gsm call <number>
```

For example, to simulate an incoming call from the number 555-1212, you would issue the following console command:

```
gsm call 5551212
```

The result of this command in the emulator is shown in Figure A.10. The name “Anne Droid” shows up because we have an entry in the Contacts database that ties the phone number 555-1212 to a contact named Anne Droid.

### Using the Console to Simulate SMS Messages

You can simulate SMS messages to the emulator from specified numbers as well, just as you can from DDMS. The command for issuing an incoming SMS is

```
sms send <number> <message>
```

For example, to simulate an incoming SMS from the number 555-1212, you would issue the following command:

```
sms send 5551212 Got Chickpeas?
```

In the emulator, you get a notification on the status bar informing you of a new message. It even displays the contents on the bar for a moment and then rolls away, showing the Message icon. You can pull down the notification bar to see the new message or launch the Messaging application. The result of the preceding command in the emulator is shown in Figure A.11.

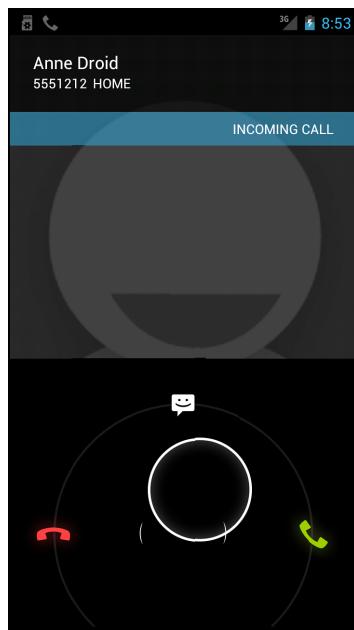


Figure A.10 Incoming call from 555-1212 (configured as a contact named Anne Droid), prompted via the Emulator console.

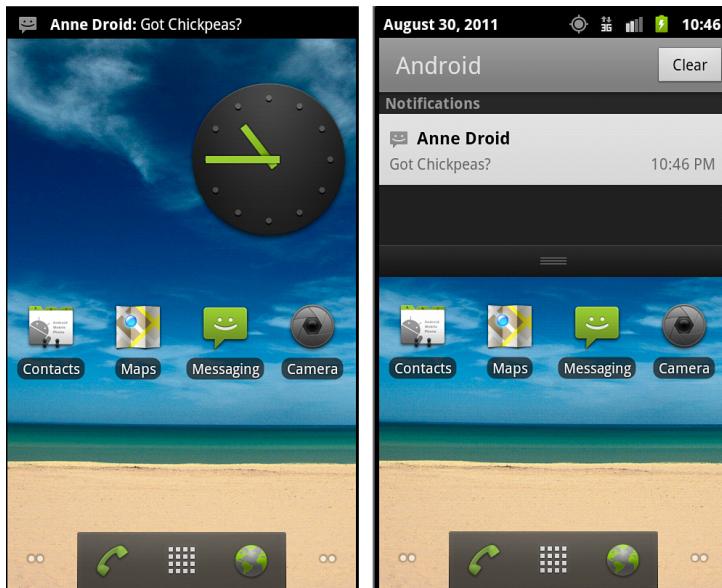


Figure A.11 An incoming SMS from 555-1212 (configured as a contact named Anne Droid), prompted via the Emulator console.

## Using the Console to Send GPS Coordinates

You can use the Emulator console to issue commands to the emulator. The command for a simple GPS fix is

```
geo fix <longitude> <latitude> [<altitude>]
```

For instance, to set the fix for the emulator to the top of Mount Everest, launch the Maps application in the emulator by selecting Menu, My Location. Then, within the Emulator console, issue the following command to set the device's coordinates appropriately:

```
geo fix 86.929837 27.99003 8850
```

## Using the Console to Monitor Network Status

You can monitor the network status of the emulator and change the network speed and latency on the fly. The command for displaying network status is

```
network status
```

Typical results from this request look something like this:

```
network status
Current network status:
download speed:      0 bits/s (0.0 KB/s)
upload speed:        0 bits/s (0.0 KB/s)
minimum latency: 0 ms
maximum latency: 0 ms
OK
```

## Using the Console to Manipulate Power Settings

You can manage “fake” power settings on the emulator using the power commands. You can turn the battery capacity to 99% charged as follows:

```
power capacity 99
```

You can turn the AC charging state to off (or on) as follows:

```
power ac off
```

You can turn the Battery status to the option unknown, charging, discharging, not-charging, or full as follows:

```
power status full
```

You can turn the Battery Present state to true (or false) as follows:

```
power present true
```

You can turn the Battery health state to the options unknown, good, overheat, dead, overvoltage, or failure as follows:

```
power health good
```

You can show the current power settings by issuing the following command:

```
power display
```

Typical results from this request look something like this:

```
power display
AC: offline
status: Full
health: Good
present: true
capacity: 99
OK
```

## Using Other Console Commands

There are also commands for simulating hardware events, port redirection, checking, starting, and stopping the virtual machine. For example, quality assurance personnel will want to check out the event subcommands, which can be used to simulate key events for automation purposes. It's likely this is the same interface used by the ADB Exerciser Monkey, which presses random keys and tries to crash your application.

## Enjoying the Emulator

Here are a few more tips for using the emulator, just for fun:

- On the Home screen, press and hold the screen to change the wallpaper and add applications, shortcuts, and widgets to the screen.
- If you press and hold an icon (usually an application icon) in the application tray, you can place a shortcut to it on your Home screen for easy access. Newer platform versions also enable other options, such as uninstalling the application or getting more information, which is very handy.
- If you press and hold an icon on your Home screen, you can move it around or dump it into the trash to get it off the screen.
- Press and fling the device's Home screen to the left and right for more space. Depending on which version of Android you're running, you find a number of other pages, with app widgets such as Google Search and lots of empty space where you can place other Home screen items.
- Another way to change your wallpaper and add applications to your Home screen is to press Menu on the screen and then choose Add. Here you can also add shortcuts and picture frame widgets around your family photo, and so on, as shown in Figure A.12.



Figure A.12 Customizing the emulator Home screen with app widgets.

In other words, the emulator can be personalized in many of the same ways as a regular device. Making these sorts of changes can be useful for comprehensive application testing.

## Understanding Emulator Limitations

The emulator is powerful, but it has several important limitations:

- It is not a device, so it does not reflect actual behavior, only simulated behavior. Simulated behavior is generally more consistent (less random) than what users experience in real life on real devices.
- It simulates phone calls and messaging, but you cannot place or receive true calls or SMS messages. No support for MMS.
- Limited ability to determine device state (network state, battery charge).
- Limited ability to simulate peripherals (camera/video capture, headphones, sensor data).
- Limited API support (for example, no SIP, hardware acceleration, OpenGL ES 2.0, and third-party hardware API support). When developing certain categories of applications, such as augmented reality applications, 3D games, and applications that rely upon sensor data, you're better off using the real hardware.

- Limited performance (modern devices often perform much better than the emulator at many tasks, such as video and animation).
- Limited support for manufacturer or operator-specific device characteristics, themes, or user experiences. Some manufacturers, such as Motorola, have provided emulator add-ons to more closely mimic the behavior of specific devices.
- On Android 4.0 and later, the emulator can use attached web cameras to emulate device hardware cameras. On previous versions of the tools, the camera would respond, but took fake pictures.
- No USB or Bluetooth support.

## References and More Information

Android Dev Guide: “Managing Virtual Devices”

<http://d.android.com/guide/developing/devices/>

Android Dev Guide: “Managing AVDs with AVD Manager”:

<http://d.android.com/guide/developing/devices/managing-avds.html>

Android Dev Guide: “Managing AVDs from the Command Line”:

<http://d.android.com/guide/developing/devices/managing-avds-cmdline.html>

Android Tools: “Emulator”:

<http://d.android.com/guide/developing/tools/emulator.html>

Android Dev Guide: “Using the Android Emulator”:

<http://d.android.com/guide/developing/devices/emulator.html>

Android Tools: “android”:

<http://d.android.com/guide/developing/tools/android.html>

*This page intentionally left blank*

# B

## The Android DDMS Quick-Start Guide

The Dalvik Debug Monitor Server (DDMS) is a debugging tool provided with the Android Software Development Kit (SDK). Developers use DDMS to provide a window into the emulator or the actual device for debugging purposes as well as file and process management. It's a blend of several tools: a task manager, a profiler, a file explorer, an emulator console, and a logging console. This Quick-Start Guide is not complete documentation of the DDMS functionality. Instead, it is designed to get you up and running with common tasks. See the DDMS documentation provided with the Android SDK for a complete list of features.

### Using DDMS with Eclipse and as a Standalone Application

If you use Eclipse with the Android Development Tools plug-in, the DDMS tool is tightly integrated with your development environment as a perspective. By using the DDMS perspective (shown in Figure B.1, using the File Explorer to browse files on the emulator instance), you can explore any emulator instances running on the development machine and any Android devices connected via USB.

If you're not using Eclipse, the DDMS tool is also available within the `/tools` directory of the Android SDK and you can launch it as a separate application, in which case it runs in its own process.

#### Tip

There should be only one instance of the DDMS tool running at a given time. This includes the Eclipse perspective. Other DDMS launches are ignored; if you have Eclipse running and try to launch DDMS from the command line, you might see question marks instead of process names, and you see debug output stating that the instance of DDMS is being ignored.

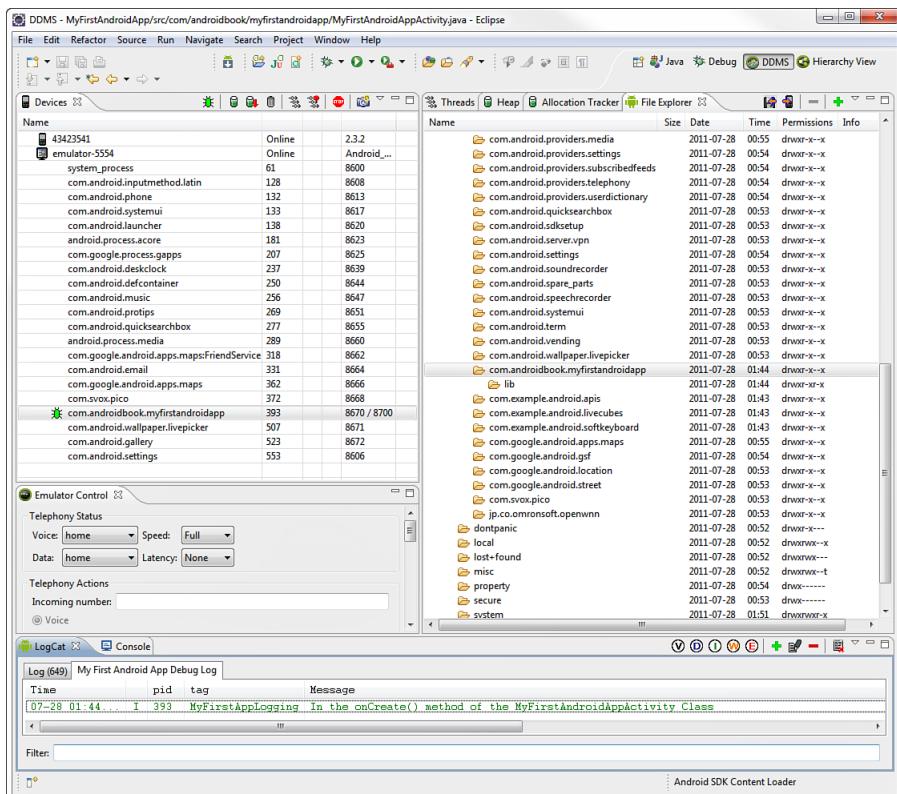


Figure B.1 The Eclipse DDMS perspective with one emulator and one Android device connected in the Devices pane.



### Warning

Not all DDMS features are available for both emulators and devices. Certain features are only available for emulators. For example, the Emulator control features are only available for use with emulators. Most devices are more secure than the emulator. As such, the File Explorer may be limited to just public areas of the device, unlike on the emulator.

## Getting Up to Speed Using Key Features of DDMS

Whether you use DDMS from Eclipse or as a standalone tool, be aware of a few key features:

- The **Devices** pane displays running emulators and connected devices in the top-left corner.

- The set of **Threads**, **Heap**, **Allocation Tracker**, and **File Explorer** tabs on the right side are populated with data when a specific process on an emulator or device is highlighted in the Devices pane.
- The **Emulator Control** pane provides features such as the ability to send GPS information and to simulate incoming calls and SMS messages to emulators.
- The **LogCat** window enables you to monitor the output of the Android logging console for a given device or emulator. This is where calls to `Log.i()`, `Log.e()`, and other log messages display.

Now let's look at how to use each of these DDMS features in more detail.



### Tip

Recently, another Eclipse perspective was added as part of the ADT plug-in to provide direct access to the Hierarchy Viewer tool, which can be used for debugging and performance-tuning your application user interface. See Chapter 4, “Mastering the Android Development Tools,” for more details about this tool.

## Working with Processes, Threads, and the Heap

One of the most useful features of DDMS is the ability to interact with processes. Each Android application runs in its own VM with its own user ID on the operating system. Using the Devices pane of DDMS, you can browse all instances of the VM running on a device, each identified by its package name. For example, you can perform the following tasks:

- Attach and debug applications in Eclipse
- Monitor threads
- Monitor the heap
- Stop processes
- Force garbage collection (GC)

## Attaching a Debugger to an Android Application

Although you'll use the Eclipse debug configurations to launch and debug your applications most of the time, you can also use DDMS to choose which application to debug and attach directly. To attach a debugger to a process, you need to have the package source code open in your Eclipse workspace. Now perform the following steps to debug:

1. On the emulator or device, verify that the application you want to debug is running.
2. In DDMS, find that application's package name in the Devices pane and highlight it.
3. Click the little green bug button ( ) to debug that application.
4. Switch to the Debug perspective of Eclipse as necessary; debug as you would normally.

## Stopping a Process

You can use DDMS to kill an Android application by following these steps:

1. On the emulator or device, verify that the application you want to stop is running.
2. In DDMS, find that application's package name in the Devices pane and highlight it.
3. Click the red stop sign button ( ) to stop that process.

## Monitoring Thread Activity of an Android Application

You can use DDMS to monitor thread activity of an individual Android application by following these steps:

1. On the emulator or device, verify that the application you want to monitor is running.
2. In DDMS, find that application's package name in the Devices pane and highlight it.
3. Click the three black arrows button ( ) to display the threads of that application. They appear in the right portion of the Threads pane. This data updates every 4 seconds by default.
4. On the Threads pane, you can choose a specific thread and click the Refresh button to drill down within that thread. The resulting classes in use display below.

For example, in Figure B.2, we see the Threads pane contents for the package named com.androidbook.myfirstandroidapp running on the emulator.



### Note

You can also start thread profiling using the button with three black arrows and a red dot ( ).

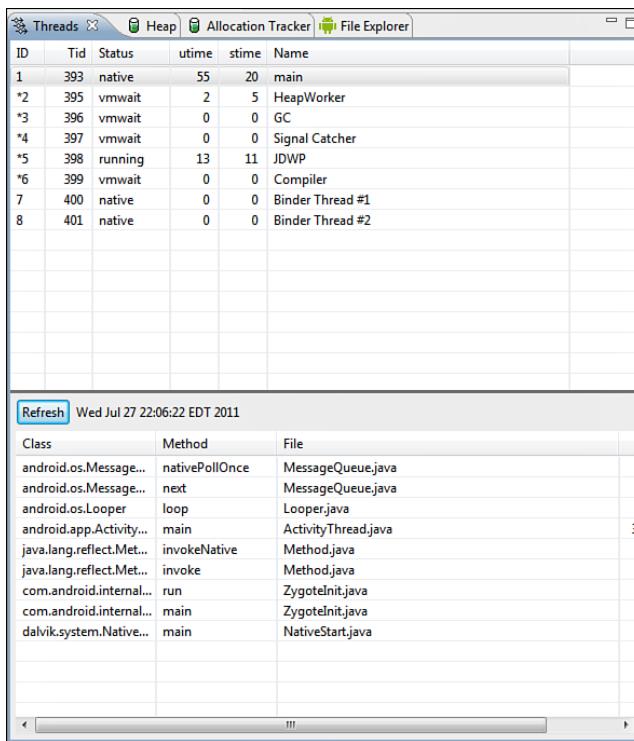


Figure B.2 Using the DDMS Threads pane.

## Monitoring Heap Activity

You can use DDMS to monitor heap statistics of an individual Android application. The heap statistics are updated after every GC via these steps:

1. On the emulator or device, verify that the application you want to monitor is running.
2. In DDMS, find that application's package name in the Devices pane and highlight it.
3. Click the green cylinder button ( ) to display the heap information for that application. The statistics appear in the Heap pane. This data updates after every GC. You can also cause GC operations from the Heap pane using the button Cause GC.
4. On the Heap pane, you can choose a specific type of object. The resulting graph in use displays at the bottom of the Heap pane, as shown in Figure B.3.

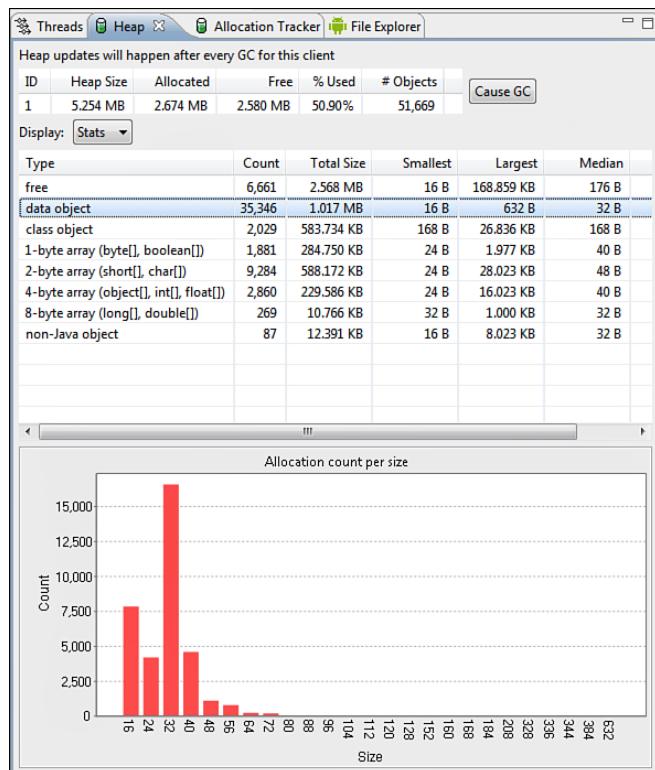


Figure B.3 Using the DDMS Heap pane.

**Tip**

When using the allocation tracker and heap monitor, keep in mind that not all memory your app uses will be accounted for in this view. This tool shows the allocations within the Dalvik VM. Some calls allocate memory on the native heap. For example, many image-manipulation calls in the SDK will result in memory allocated natively and not show up in this view.

## Prompting Garbage Collection

You can use DDMS to force the garbage collection (GC) to run by following these steps:

1. On the emulator or device, verify that the application you want to run GC for is running.
2. In DDMS, find that application's package name in the Devices pane and highlight it.

- Click the garbage can button ( ) to cause garbage collection to run for the application. The results can be viewed in the Heap pane.

## Creating and Using an HPROF File

HPROF files can be used to inspect the heap and are used for profiling and performance purposes. You can use DDMS to create an HPROF file for your application to be created by following these steps:

- On the emulator or device, verify that the application you want the HPROF data for is running.
- In DDMS, find that application's package name in the Devices pane and highlight it.
- Click the HPROF button ( ) to create an HPROF dump to be generated for the application. The files will be generated in the /data/misc/ directory.

For example, in Figure B.4, you can see the HPROF dump response in Eclipse. You are switched to the Debug perspective, and a graphical trace is displayed.

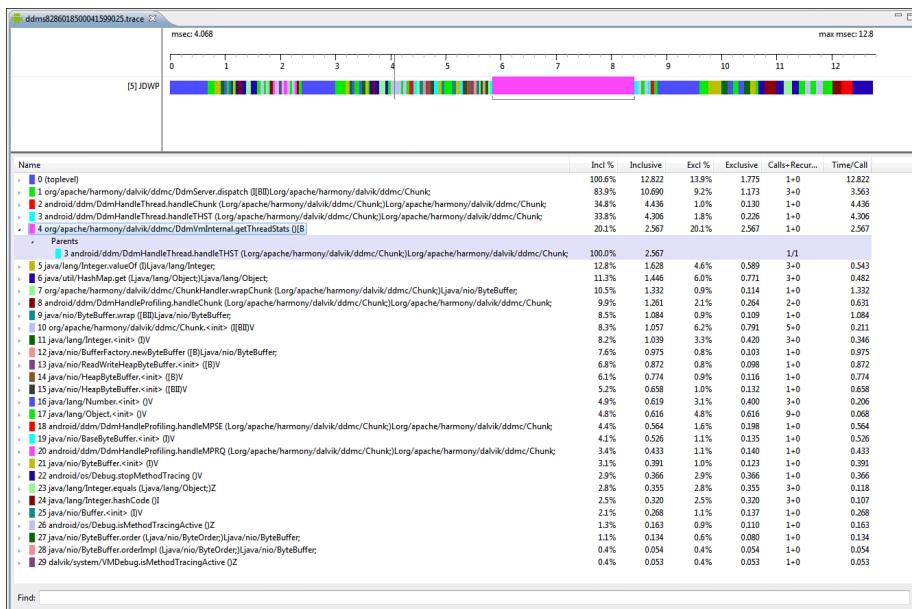


Figure B.4 Using Eclipse to inspect HPROF profiling information.

Once you have Android-generated HPROF data, you can convert it to a standard HPROF file format using the Android SDK tool called `hprof-conv`. You can use whichever profiling tool you prefer to examine the information.



### Note

You can generate HPROF files in Android using several other methods. For example, you can do it programmatically using the `Debug` class. The `monkey` tool also has options for generating HPROF files as it runs.

## Using the Allocation Tracker

You can use DDMS to monitor memory allocated by a specific Android application. The memory allocation statistics are updated on demand by the developer. Follow these steps to track memory allocations:

1. On the emulator or device, verify that the application you want to monitor is running.
2. In DDMS, find that application's package name in the Devices pane and highlight it.
3. Switch to the Allocation Tracker pane on the right pane.
4. Click the Start Tracking button to start tracking memory allocations and the Get Allocations button to get the allocations at a given time.
5. To stop tracking allocations, click the Stop Tracking button.

For example, in Figure B.5, we see the Allocation Tracker pane contents for an application running on the emulator.

The Android developer website has a write-up on how to track memory allocations at <http://goo.gl/wR8Oa> and further information on memory analysis at <http://goo.gl/turJZ>.

## Working with the File Explorer

You can use DDMS to browse and interact with the Android file system on an emulator or device (although it's somewhat limited on devices without root access). You can access application files, directories, and databases, as well as pull and push files to the Android system, provided you have the appropriate permissions.

For example, in Figure B.6, we see the File Explorer pane contents for the emulator.

The screenshot shows the DDMS Allocation Tracker pane. At the top, there are tabs for Threads, Heap, Allocation Tracker, and File Explorer. Below the tabs are buttons for Stop Tracking, Get Allocations, Filter, and Inc. trace. The main area contains two tables. The first table, titled 'Alloc Order', lists allocations by address (Alloc Order), Allocated Class (e.g., byte[], char[]), Thread Id (e.g., 5), and Allocated in (e.g., org.apache.harmony.dalvik.ddmc.Dd...). The second table, titled 'Class', lists methods and their file locations and line numbers.

Alloc Order	Allocated Class	Thread Id	Allocated in	Allocated in
50	164 byte[]	5	org.apache.harmony.dalvik.ddmc.Dd...	getThreadStats
44	164 byte[]	5	org.apache.harmony.dalvik.ddmc.Dd...	getThreadStats
38	164 byte[]	5	org.apache.harmony.dalvik.ddmc.Dd...	getThreadStats
24	164 byte[]	5	org.apache.harmony.dalvik.ddmc.Dd...	getThreadStats
18	164 byte[]	5	org.apache.harmony.dalvik.ddmc.Dd...	getThreadStats
1	68 char[]	5	android.ddm.DdmHandleHeap	handleREAL
8	46 char[]	5	android.ddm.DdmHandleHeap	handleIPGC
51	44 java.nio.ReadWriteH...	5	java.nio.BufferFactory	newByteBuffer
45	44 java.nio.ReadWriteH...	5	java.nio.BufferFactory	newByteBuffer
39	44 java.nio.ReadWriteH...	5	java.nio.BufferFactory	newByteBuffer
..	..	..	..	..

Class	Method	File	Line	Native
org.apache.harmony...	getThreadStats	DdmVmInternal.java	-2	true
android.ddm.DdmH...	handleTHST	DdmHandleThread.java	107	false
android.ddm.DdmH...	handleChunk	DdmHandleThread.java	76	false
org.apache.harmony...	dispatch	DdmServer.java	171	false
dalvik.system.Native...	run	NativeStart.java	-2	true
..	..	..	..	..

Figure B.5 Using the DDMS Allocation Tracker pane.

The screenshot shows the DDMS File Explorer pane. At the top, there are tabs for Threads, Heap, Allocation Tracker, and File Explorer. The main area displays a file tree under the 'data' directory. A specific file, 'com.androidbook.myfirstandroidapp', is selected and highlighted with a gray background. The tree includes various sub-directories like lib, com.example.android.apis, and com.google.android.maps, along with system files such as lost+found and property.

Name	Size	Date	Time	Permissions	Info
app-private		2011-07-28	00:52	drwxrwx--x	
backup		2011-07-28	00:54	drwx-----	
dalvik-cache		2011-07-28	01:50	drwxrwx--x	
data		2011-07-28	01:22	drwxrwx--x	
android.tts		2011-07-28	00:53	drwxr-x--x	
com.android.browser		2011-07-28	00:54	drwxr-x--x	
com.android.sdksetup		2011-07-28	00:53	drwxr-x--x	
com.android.server.vpn		2011-07-28	00:53	drwxr-x--x	
com.android.settings		2011-07-28	00:54	drwxr-x--x	
com.android.soundrecorder		2011-07-28	00:53	drwxr-x--x	
com.android.spare_parts		2011-07-28	00:53	drwxr-x--x	
com.android.speechrecorder		2011-07-28	00:53	drwxr-x--x	
com.android.systemui		2011-07-28	00:53	drwxr-x--x	
com.android.term		2011-07-28	00:53	drwxr-x--x	
com.android.vending		2011-07-28	00:53	drwxr-x--x	
com.android.wallpaper.livewallpaper		2011-07-28	00:53	drwxr-x--x	
com.androidbook.myfirstandroidapp		2011-07-28	01:44	drwxr-x--x	
lib		2011-07-28	01:44	drwxr-x--x	
com.example.android.apis		2011-07-28	01:43	drwxr-x--x	
com.example.android.liveweb		2011-07-28	01:43	drwxr-x--x	
com.example.android.softkeyboard		2011-07-28	01:43	drwxr-x--x	
com.google.android.apps.maps		2011-07-28	00:55	drwxr-x--x	
com.google.android.gsf		2011-07-28	00:54	drwxr-x--x	
com.google.android.location		2011-07-28	00:53	drwxr-x--x	
com.google.android.street		2011-07-28	00:53	drwxr-x--x	
com.svox.pico		2011-07-28	00:53	drwxr-x--x	
jp.co.omronsoft.openwnn		2011-07-28	00:53	drwxr-x--x	
dontpanic		2011-07-28	00:52	drwxr-x---	
local		2011-07-28	00:52	drwxrwx--x	
lost+found		2011-07-28	00:52	drwxrwx---	
misc		2011-07-28	00:52	drwxrwx-t	
property		2011-07-28	00:54	drwx-----	

Figure B.6 Using the DDMS File Explorer pane.

## Browsing the File System of an Emulator or Device

To browse the Android file system, follow these steps:

1. In DDMS, choose the emulator or device you want to browse in the Devices pane.
2. Switch to the File Explorer pane. You see a directory hierarchy.
3. Browse to a directory or file location.

Keep in mind that directory listings in the File Explorer might take a moment to update when contents change.



### Note

Some device directories, such as the /data directory, might not be accessible from the DDMS File Explorer.

Table B.1 shows some important areas of the Android file system. Although the exact directories may vary from device to device, the directories listed are the most common.

**Table B.1 Important Directories in the Android File System**

Directory	Purpose
/data/app/	Where Android APK files are stored.
/data/data/<package name>/	Application top-level directory; for example: /data/data/com.androidbook.pettracker/.
/data/data/<package name>/shared_prefs/	Application shared preferences directory. Named preferences are stored as XML files.
/data/data/<package name>/files/	Application file directory.
/data/data/<package name>/cache/	Application cache directory.
/data/data/<package name>/databases/	Application database directory; for example: /data/data/com.androidbook.pettracker/databases/test.db.
/mnt/sdcard/	External storage (SD card).
/mnt/sdcard/download/	Where browser images are saved.

## Copying Files from the Emulator or Device

You can use File Explorer to copy files or directories from an emulator or a device file system to your computer by following these steps:

1. Using File Explorer, browse to the file or directory to copy and highlight it.
2. From the top-right corner of the File Explorer, click the Disk button with the arrow (⬇) to pull the file from the device. Alternatively, you can pull down the drop-down menu next to the buttons and choose Pull File.
3. Type in the path where you want to save the file or directory on your computer and click Save.

## Copying Files to the Emulator or Device

You can use File Explorer to copy files to an emulator or a device file system from your computer by following these steps:

1. Using File Explorer, browse to the file or directory to copy and highlight it.
2. From the top-right corner of File Explorer, click the Phone button with the arrow (⬆) to push a file to the device. Alternatively, you can pull down the drop-down menu next to the buttons and choose Push File.
3. Select the file or directory on your computer and click Open.

### Tip

File Explorer also supports some drag-and-drop operations. This is the only way to push directories to the Android file system; however, copying directories to the Android file system is not recommended because there's no delete option for them. You need to delete directories programmatically if you have the permissions to do so. Alternately, the adb shell can be used with rmdir, but you still need permissions to do so. That said, you can drag a file or directory from your computer to File Explorer and drop it in the location you want.

## Deleting Files on the Emulator or Device

You can use File Explorer to delete files (one at a time, and not directories) on the emulator or device file system. Follow these steps:

1. Using File Explorer, browse to the file you want to delete and highlight it.
2. In the top-right corner of File Explorer, click the red minus button (⊖) to delete the file.

### Warning

Be careful. There is no confirmation. The file is deleted immediately and is not recoverable.

## Working with the Emulator Control

You can use DDMS to interact with instances of the emulator using the Emulator Control pane. You must select the emulator you want to interact with for the Emulator Control pane to work. You can use the Emulator Control pane to do the following:

- Change telephony status
- Simulate incoming voice calls
- Simulate incoming SMS messages
- Send a location fix (GPS coordinates)

### Simulating Incoming Voice Calls

To simulate an incoming voice call using the Emulator Control pane, use the following steps:

1. In DDMS, choose the emulator you want to call in the Devices pane.
2. Switch to the Emulator pane. You work with the Telephony Actions.
3. Input the incoming phone number. This might include only numbers, +, and #.
4. Select the Voice radio button.
5. Click the Call button.
6. In the emulator, your phone is ringing. Answer the call.
7. The emulator can end the call as normal, or you can end the call in DDMS using the Hang Up button.

### Simulating Incoming SMS Messages

DDMS provides the most stable method to send incoming SMS messages to the emulator. You send an SMS much as you initiated the voice call. To simulate an incoming SMS message using the Emulator Control pane, use the following steps:

1. In DDMS, choose the emulator you want to send a message to in the Devices pane.
2. Switch to the Emulator pane. You work with the Telephony Actions.
3. Input the incoming phone number. This might include only numbers, +, and #.
4. Select the SMS radio button.
5. Type in your SMS message.
6. Click the Send button.
7. Over in the emulator, you receive an SMS notification.

## Sending a Location Fix

The steps for sending GPS coordinates to the emulator are covered in Appendix A, “The Android Emulator Quick-Start Guide.” Simply input the GPS information into the Emulator Control pane, click Send, and use the Maps application on the emulator to get the current position.

## Taking Screen Captures of the Emulator and Device Screens

You can take screen captures of the emulator and the device from DDMS. The device captures are most useful for debugging, and this makes the DDMS tool appropriate for quality assurance personnel and developers. To capture a screenshot, take the following steps:

1. In DDMS, choose the emulator or device you want to take a capture of in the Devices pane.
2. On the device or emulator, make sure you have the screen you want to capture.
3. Click the multicolored square picture button ( ) to take a screen capture. A capture window launches, as shown in Figure B.7.

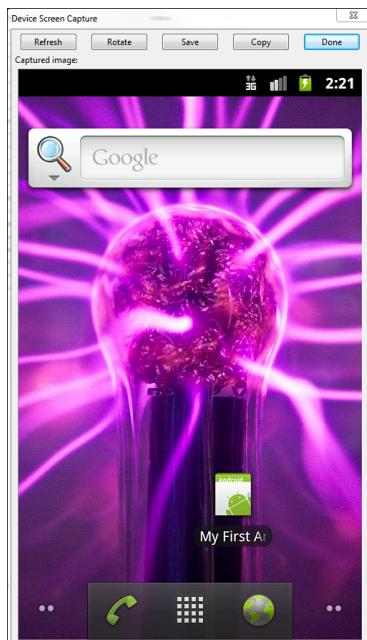


Figure B.7 Using DDMS to take a screenshot.

4. Within the capture window, click the Save button to save the screen capture. Similarly, the Copy button stores the screenshot in your clipboard and the Refresh button updates the screenshot if the underlying device or emulator screen has changed since you launched the capture window.

## Working with Application Logging

The LogCat tool is integrated into DDMS. It is provided as a pane along the bottom of the DDMS user interface. You can control how much information displays by clicking the little round circles with letters in them. The V stands for verbose (show everything) and is the default. The other options correspond to Debug (D), Information (I), Warning (W), and Error (E). When selected, only log entries for that level of severity and worse will display. You can filter the LogCat results to show just search results by using the search entry field, which fully supports regular expressions. Search terms can be limited in scope with prefixes, such as “text:” to limit the following term to just the log message text.

You can also create saved filters to display only the LogCat information associated with particular attributes. You can use the plus (+) button to add a saved filter and show only log entries matching a tag, message, process ID, name, or log level. The strings for each attribute filter can also be Java-style regular expressions.

For example, suppose your application does this:

```
public static final String DEBUG_TAG = "MyFirstAppLogging";
Log.i(DEBUG_TAG,
    "In the onCreate() method of the MyFirstAndroidAppActivity Class.");
```

You can create a LogCat filter using the plus button (). Name the filter and set the log tag to the string matching your debug tag:

MyFirstAppLogging

The LogCat pane with the resulting filter is shown in Figure B.8.

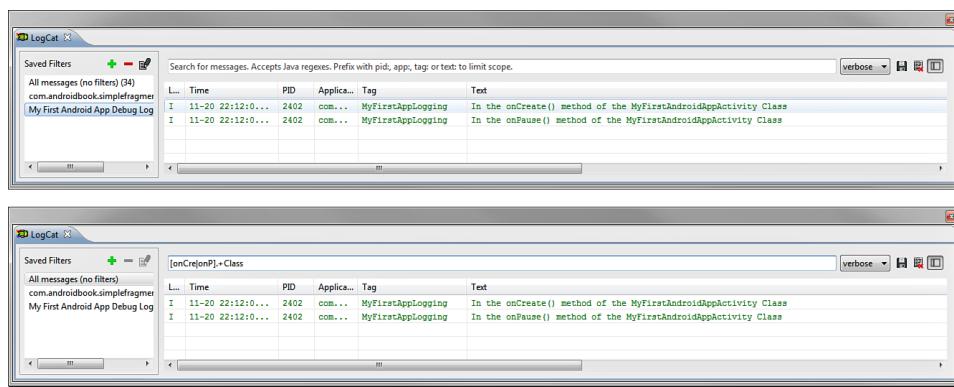


Figure B.8 Using the DDMS LogCat logging pane with a custom filter (top) and a regular expression search (bottom).

To search for the message if you don't need to create a full tab, you could type in "MyFirst" and get all the results with tags and text fields containing this string.

*This page intentionally left blank*

# C

## Eclipse IDE Tips and Tricks

The Eclipse IDE is the most popular development environment for Android developers. In this appendix, we provide a number of helpful tips and tricks for using Eclipse to develop Android applications quickly and effectively.



### Note

Do you have your own tips or tricks for Android development in Eclipse? If so, email them to us (with permission to publish them) at [androidwirelessdev@gmail.com](mailto:androidwirelessdev@gmail.com), and they might be included on our blog at <http://androidbook.blogspot.com> or the next edition of one of our books. Get your moment of geekly fame!

## Organizing Your Eclipse Workspace

In this section, we provide a number of tips and tricks to help you organize your Eclipse workspace for optimum Android development.

### Integrating with Source Control Services

Eclipse has the ability to integrate with many source control packages using add-ons or plug-ins. This allows Eclipse to manage checking out a file (making it writable) when you first start to edit a file, checking a file in, updating a file, showing a file's status, and a number of other tasks, depending on the support of the add-on.



### Tip

Common source control add-ons are available for CVS, Subversion, Perforce, git, Mercurial, and many other packages.

Generally speaking, not all files are suitable for source control. For Android projects, any file within the `/bin` and `/gen` directories shouldn't be in source control. To exclude these generically within Eclipse, go to Preferences, Team, Ignored Resources. You can add file suffixes such as `*.apk`, `*.ap_`, and `*.dex` by clicking the Add Pattern button

and adding one at a time. Conveniently, this applies to all integrated source control systems.

## Repositioning Tabs within Perspectives

Eclipse provides some pretty decent layouts with the default perspectives. However, not every one works the same way. We feel that some of the perspectives have poor default layouts for Android development and could use some improvement.



### Tip

Experiment to find a tab layout that works well for you. Each perspective has its own layout, too, and the perspectives can be task oriented.

For instance, the Properties tab is usually found on the bottom of a perspective. For code, this works fine because this tab is only a few lines high. But for resource editing in Android, this doesn't work so well. Luckily, in Eclipse, this is easy to fix: Simply drag the tab by left-clicking and holding on the tab (the title) itself and dragging it to a new location, such as the vertical section on the right side of the Eclipse window. This provides the much-needed vertical space to see the dozens of properties often found here.



### Tip

If you mess up a perspective or just want to start fresh, you can reset it by choosing Window, Reset Perspective.

## Maximizing Windows

Sometimes, you might find that the editor window is just too small, especially with all the extra little metadata windows and tabs surrounding it. Try this: Double-click the tab of the source file that you want to edit. Boom! It's now nearly the full Eclipse window size! Just double-click to return it to normal. (Ctrl+M works on Windows, Command+M on the Mac.)

## Minimizing Windows

You can minimize entire sections, too. For instance, if you don't need the section at the bottom that usually has the console or the one to the left that usually has the File Explorer view, you can use the minimize button in each section's upper-right corner. Use the button that looks like two little windows to restore it.

## Viewing Windows, Side by Side

Ever wish you could see two source files at once? Well, you can! Simply grab the tab for a source file and drag it either to the edge of the editor area or to the bottom. You then

see a dark outline, showing where the file will be docked—either side-by-side with another file or above or below another file. This creates a parallel editor area where you can drag other file tabs as well. You can repeat this multiple times to show three, four, or more files at once.

## **Viewing Two Sections of the Same File**

Ever wish you could see two places at once in the same source file? You can! Right-click the tab for the file in question and choose New Editor. A second editor tab for the same file comes up. With the previous tip, you can now have two different views of the same file.

## **Closing Unwanted Tabs**

Ever feel like you get far too many tabs open for files you’re no longer editing? I do! There are a number of solutions to this problem. First, you can right-click a file tab and choose Close Others to close all other open files. You can quickly close specific tabs by middle-clicking each tab. (This even works on a Mac with a mouse that can middle click, such as one with a scroll wheel.)

## **Keeping Windows Under Control**

Finally, you can use the Eclipse setting that limits the number of open file editors:

1. Open Eclipse’s Preferences dialog.
2. Expand General, choose Editors, and check Close Editors Automatically.
3. Edit the value in Number of Opened Editors Before Closing.

This will cause old editor windows to be closed when new ones are open. Eight seems to be a good number to use for the Number of Opened Editors Before Closing option to keep the clutter down but to have enough editors open to still get work done and have reference code open. Note also that if you check Open New Editor under When All Editors Are Dirty or Pinned, more files will be open if you’re actively editing more than the number chosen. Thus, this setting doesn’t affect productivity when you’re editing a large number of files all at once but can keep things clean during most normal tasks.

## **Creating Custom Log Filters**

Every Android log statement includes a tag. You can use these tags with filters defined in LogCat. To add a new filter, click the green plus sign button in the LogCat pane. Name the filter—perhaps using the tag name—and fill in the tag you want to use. Now there is another tab in LogCat that shows messages that contain this tag. In addition, you can create filters that display items by severity level.

Android convention has largely settled on creating tags based on the name of the class. You see this frequently in the code provided with this book. Note that we create a constant in each class with the same variable name to simplify each logging call. Here's an example:

```
public static final String DEBUG_TAG = "MyClassName";
```

This convention isn't a requirement, though. You could organize tags around specific tasks that span many activities or you could use any other logical organization that works for your needs. Another simpler way to do this is as follows:

```
private final String DEBUG_TAG = getClass().getSimpleName();
```

Although not as efficient at runtime, this code can help you avoid copy and paste errors. If you've ever been looking over a log file and had a misnamed debug tag string mislead you, this trick may be useful to you.

## Searching Your Project

You have several ways to easily search your project files from within Eclipse. The search options are found under the Search menu of the Eclipse toolbar. Most frequently, we use the File Search option, which allows you to search for text within the files found in the workspace, as well as files by name. The Java Search option can also help you find Java-specific elements of your project such as methods and fields.

## Organizing Eclipse Tasks

By default, any comment that starts with // TODO will show up on the Tasks tab in the Java perspective. This can be helpful for tagging code areas that require further implementation. You can click a specific task and it will take you straight to the comment in the file so you can implement the item at a later time.

You can also create custom comment tags above and beyond To-Do items. We often leave comments with a person's initials to make it easy for them to find specific functional areas of the application to code review. Here's an example:

```
// LED: Does this look right to you?  
// CDC: Related to Bug 1234. Can you fix this?  
// SAC: This will have to be incremented for the next build
```

You might also use a special comment such as //HACK: when you have to implement something that is less than an ideal implementation, to flag that code as subject to further review. To add custom tags to your Task list, edit your Eclipse preferences (available at Window, Preferences) and navigate to Java, Compiler, Task Tags. Add any tags you want to flag. The tags can be flagged for a certain priority level. So, for instance, something with your initials might be high priority to look at right away, but a HACK flag may be low priority because, presumably, it works but maybe not in the best way possible.

# Writing Code in Java

In this section, we provide a number of tips and tricks to help you implement the code for your Android applications.

## Using Autocomplete

Autocomplete is a great feature that speeds up code entry. If this feature hasn't appeared for you yet or has gone away, you can bring it up by pressing Ctrl+spacebar. Autocomplete not only saves time in typing but can be used to jog your memory about methods—or to help you find a new method. You can scroll through all the methods of a class and even see the Javadocs associated with them. You can easily find static methods by using the class name or the instance variable name. You follow the class or variable name with a dot (and maybe Ctrl+spacebar) and then scroll through all the names. Then you can start typing the first part of a name to filter the results.

## Creating New Classes and Methods

You can quickly create a new class and corresponding source file by right-clicking the package to create it and then choosing New, Class. Next, you enter the class name, pick a superclass and interfaces, and choose whether to create default comments and method stubs for the superclass for constructors or abstract methods.

Along the same lines as creating new classes, you can quickly create method stubs by right-clicking a class or within a class in the editor and choosing Source, Override/Implement Methods. Then you choose the methods for which you're creating stubs, where to create the stubs, and whether to generate default comment blocks.

## Organizing Imports

When referencing a class in your code for the first time, you can hover over the newly used class name and choose Import “*Classname*” (*package name*) to have Eclipse quickly add the proper import statement.

In addition, the Organize Imports command (Ctrl+Shift+O in Windows and Command+Shift+O on a Mac) causes Eclipse to automatically organize your imports. Eclipse removes unused imports and adds new ones for packages used but not already imported.

If there is any ambiguity in the name of a class during automatic import, such as with the Android Log class, Eclipse prompts you with the package to import. Finally, you can configure Eclipse to automatically organize the imports each time you save a file. This can be set for the entire workspace or for an individual project.

Configuring this for an individual project gives you better flexibility when you're working on multiple projects and don't want to make changes to some code, even if the changes are an improvement. To configure this, perform the following steps:

1. Right-click the project and choose Properties.
2. Expand Java Editor and choose Save Actions.

3. Check Enable Project Specific Settings, Perform the Selected Actions on Save, and Organize Imports.

## Formatting Code

Eclipse has a built-in mechanism for formatting Java code. Formatting code with a tool is useful for keeping the style consistent, applying a new style to old code, or matching styles with a different client or target (such as a book or an article).

To quickly format a small block of code, select the code and press Ctrl+Shift+F in Windows (or Command+Shift+F on a Mac). The code is formatted to the current settings. If no code is selected, the entire file is formatted. Occasionally, you need to select more code—such as an entire method—to get the indentation levels and brace matching correct.

The Eclipse formatting settings are found in the Properties pane under Java Code Style, Formatter. You can configure these settings on a per-project or workspace-wide basis. You can apply and modify dozens of rules to suit your own style.

## Renaming Almost Anything

Eclipse’s Rename tool is quite powerful. You can use it to rename variables, methods, class names, and more. Most often, you can simply right-click the item you want to rename and then choose Refactor, Rename. Alternatively, after selecting the item, you can press Ctrl+Alt+R in Windows (or Cmd+Alt+R on a Mac) to begin the renaming process. If you are renaming a top-level class in a file, the filename has to be changed as well. Eclipse usually handles the source control changes required to do this, if the file is being tracked by source control. If Eclipse can determine that the item is in reference to the identically named item being renamed, all instances of the name are renamed as well. Occasionally, this even means comments are updated with the new name. Quite handy!

## Refactoring Code

Do you find yourself writing a whole bunch of repeating sections of code that look, for instance, like the following?

```
TextView nameCol = new TextView(this);
nameCol.setTextColor(getResources().getColor(R.color.title_color));
nameCol.setTextSize(getResources().
getDimension(R.dimen.help_text_size));
nameCol.setText(scoreUserName);
table.addView(nameCol);
```

This code sets text color, text size, and text. If you’ve written two or more blocks that look like this, your code could benefit from refactoring. Eclipse provides two useful tools—Extract Local Variable and Extract Method—to speed this task and make it almost trivial.

## Using the Extract Local Variable Tool

Follow these steps to use the Extract Local Variable tool:

1. Select the expression `getResources().getColor(R.color.title_color)`.
2. Right-click and choose Refactor, Extract Local Variable (or press `Ctrl+Alt+L`).
3. In the dialog that appears, enter a name for the variable and leave the Replace All Occurrences check box selected. Then click OK and watch the magic happen.
4. Repeat steps 1–3 for the text size.

The result should now look like this:

```
int textColor = getResources().getColor(R.color.title_color);
float textSize = getResources().getDimension(R.dimen.help_text_size);
TextView nameCol = new TextView(this);
nameCol.setTextColor(textColor);
nameCol.setTextSize(textSize);
nameCol.setText(scoreUserName);
table.addView(nameCol);
```

All repeated sections of the last five lines also have this change made. How convenient is that?

## Using the Extract Method Tool

Now you're ready for the second tool. Follow these steps to use the Extract Method tool:

1. Select all five lines of the first block of code.
2. Right-click and choose Refactor, Extract Method (or choose `Ctrl+Alt+M`).
3. Name the method and edit the variable names anything you want. (Move them up or down, too, if desired.) Then click OK and watch the magic happen.

By default, the new method is below your current one. If the other blocks of code are actually identical (meaning the statements of the other blocks must be in the exact same order), the types are all the same, and so on, they will also be replaced with calls to this new method! You can see this in the count of additional occurrences shown in the dialog for the Extract Method tool. If that count doesn't match what you expect, check that the code follows exactly the same pattern. Now you have code that looks like the following:

```
addTextToRowWithValues(newRow, scoreUserName, textColor, textSize);
```

It is easier to work with this code than with the original code, and it was created with almost no typing! If you had ten instances before refactoring, you've saved a lot of time by using a useful Eclipse feature.

## Reorganizing Code

Sometimes, formatting code isn't enough to make it clean and readable. Over the course of developing a complex activity, you might end up with a number of embedded classes and methods strewn about the file. A quick Eclipse trick comes to the rescue: With the file in question open, make sure the outline view is also visible.

Simply click and drag methods and classes around in the outline view to place them in a suitable logical order. Do you have a method that is only called from a certain class but available to all? Just drag it in to that class. This works with almost anything listed in the outline, including classes, methods, and variables.

## Using QuickFix

The QuickFix feature, accessible under Edit, Quick Fix (or Ctrl+1 in Windows and Command+1 on a Mac), isn't just for fixing possible issues. It brings up a menu of various tasks that can be performed on the highlighted code, and it shows what the change will look like. One useful QuickFix now available is the Android "Extract String" command. Use QuickFix on a string literal and you can quickly move it into an Android strings resource file, and the code is automatically updated to use the string resource. Consider how QuickFix Extract String would work on the following two lines:

```
Log.v(DEBUG_TAG, "Something happened");
String otherString = "This is a string literal.;"
```

The updated Java code is shown here:

```
Log.v(DEBUG_TAG, getString(R.string.something_happened));
String otherString = getString(R.string.string_literal);
```

And these entries have been added to the string resource file:

```
<string name="something_happened">Something happened</string>
<string name="string_literal">This is a string literal.</string>
```

The process also brings up a dialog for customizing the string name and which alternative resource file it should appear in, if any.

The QuickFix feature can be used in layout files with many Android-specific options for performing tasks such as extracting styles, extracting pieces to an include file, wrapping pieces in a new container, and even changing the widget type.

## Providing Javadoc-Style Documentation

Regular code comments are useful (when done right). Comments in Javadoc style appear in code-completion dialogs and other places, thus making them even more useful. To quickly add a Javadoc comment to a method or class, simply press Ctrl+Shift+J in Windows (or Command+Alt+J on a Mac). Alternatively, you can choose Source, Generate Element Comment to prefill certain fields in the Javadoc, such as parameter names and author, thus speeding the creation of this style of comment. Finally, if you

simply start the comment block with `/**` and press Enter, the appropriate code block will be generated and prefilled as before.

## Resolving Mysterious Build Errors

Occasionally, you might find that Eclipse is finding build errors where there were none just moments before. In such a situation, you can try a couple quick Eclipse tricks.

First, try refreshing the project: Simply right-click the project and choose Refresh or press F5. If this doesn't work, try deleting the `R.java` file, which you can find under the `gen` directory under the name of the particular package being compiled. (Don't worry: This file is created during every compile.) If the `Compile Automatically` option is enabled, the file is re-created. Otherwise, you need to compile the project again.

A second method for resolving certain build errors involves source control. If the project is managed by Eclipse via the Team, Share Project menu selection, Eclipse can manage files that are to be read-only or automatically generated. Alternatively, if you can't or don't want to use source control, make sure all of the files in the project are writeable (that is, not read-only).

Finally, you can try cleaning the project. To do this, choose Project, Clean and choose the project(s) you want to clean. Eclipse removes all temporary files and then rebuilds the project(s). If the project was an NDK project, don't forget to recompile the native code.

*This page intentionally left blank*

# Index

## Symbols

---

- # (hash symbols), **151**
- (...) (ellipsis), **174**

## A

---

- above attribute, **213**
- AbsoluteLayout class, **214**
- abstracted LCD density, **407**
- accelerometer support, **406**
- accessing
  - Booleans, **150**
  - browser information, **289–290**
  - calendar data, **291**
  - call logs, **288–289**
  - colors, **151**
  - contacts, **292**
  - content providers, **286**
  - device settings, **292**
  - dictionary, **291**
  - dimensions, **153**
  - directories, **105**
  - drawables, **154**
  - external storage, **284**
  - files, **105, 278**
  - images, **155**
  - integers, **150**
  - layouts, **166**
  - media, **286–288**

- menus, 159
- nine-patch stretchable graphics, 155
- preferences, 105, 267-268
- raw files, 161
- resources, 142
- screen information, 304
- strings, 147-148
- tweened animations, 158
- voicemail, 291
- XML files, 160
- activities**
  - application navigation, 115-116
  - avoiding killing, 109-110
  - callbacks, 107-109
    - initializing/retrieving activities, 109
    - static data, initializing, 108
    - stopping/saving/releasing, 109
  - defined, 103
  - dialogs, adding, 254
  - fragments
    - Activity classes, creating, 246-247
    - applications, designing, 238
    - attaching, 237
    - compatibility, 305
    - creating, 237
    - defined, 104, 111
    - defining, 235
    - destroying, 237
    - detaching, 237
    - dialogs, 238, 257-260
    - layout resource files, creating, 244-246
    - legacy support, 247-248
    - lifecycle, 235
    - ListFragment, implementing, 240-243
    - lists, 238
  - MP3 music player example, 111-113
  - overview, 233
  - pausing, 237
  - resuming activity, 237
  - screen workflow, 234
  - specialty, 237
  - stopping activity, 237
  - updates, 236
  - user preferences, 238
  - visibility, 237
  - web content, 238
  - website, 118
  - WebView, implementing, 243-244
  - workflow flexibility, improving, 111-113
- game application example, 106
- intents
  - action/data, 114
  - additional information, adding, 115
  - application navigation, 115-116
  - broadcasting, 117
  - filters, 114, 132-133
  - Google common, 115
  - launching activities by class name, 113
  - launching external activities, 114
  - primary entry points, choosing, 132
  - website, 118
- launch, creating, 64
- lifecycle, 106-107
- registering, 131-133
- src/com/androidbook/
  - myfirstandroidapp/MyFirstAndroid
  - AppActivity.java, 65
- stacks, 107
- state, saving into bundles, 110
- static data, destroying, 110

**Activity class**

callbacks, 107-109  
onCreate( ), 108  
onPause( ), 109  
onResume( ), 109  
defined, 103  
website, 81, 118

**ActivityTestCase class, 359**

ad-hoc permissions, 31  
ad revenue, 379

**adapters**

AdapterView class, 222-224  
arrays, 222-223  
classes, 222  
creating, 222  
databases, 223-224  
defined, 222  
lists, assigning, 225

**AdapterView class, 222-224**

ADB (Android Debug Bridge), 87, 100  
addFooterView( ) method, 226  
addHeaderView( ) method, 226

**adding**

additional activity information with intents, 115  
applications on Home screen, 419  
contacts, 297-298  
dialogs  
activities, 254  
fragment-based method, 252  
legacy method, 251  
location-based services, 76-78  
logging support, 73-74  
media support, 74, 76  
MP3 playback, 75  
pages, 419  
permissions, 75

preferences, 264-267

progress bars to title bars, 193

records, 297-298

sample projects, 52

statistics collection, 379

Support (Compatibility) Package, 248-249, 305

View controls to ViewGroup, 204

**addPreferencesFromResource( ) method, 271****addTab( ) method, 227, 229****addToBackStack( ) method, 236****addView( ) method, 204****addWord( ) method, 291****Adobe AIR website, 33****ADT Eclipse plug-in, 46-47, 88**

resources

creating, 143-146

editors, 89

layout resource editor, 164-166

UI designer, 90

**advantages**

Android, 22, 25  
application development, 26  
application integration, 26  
development tools, 25  
free market, 27-28  
IDEs, 25  
Java, 26  
open-source, 25  
publication, 27  
SDKs, 25

**AlarmClock content provider, 285, 300****alert dialogs, 252****AlertDialog class, 252, 260****aliases (resources), 162****alignBottom attribute, 213****alignLeft attribute, 212**

**alignParentBottom attribute**, 212  
**alignParentLeft attribute**, 212  
**alignParentRight attribute**, 212  
**alignParentTop attribute**, 212  
**alignRight attribute**, 212  
**alignTop attribute**, 213  
**alternative marketplaces**, 393  
**alternative resources**, 141-142

- benefits, 308
- creating, 309
- data, saving, 318
- defined, 308
- directory qualifiers, 309
  - bad examples, 311
  - case, 310
  - combining, 310
  - customizing, 311
  - default resources, including, 311
  - good examples, 311
  - limits, 310
  - listing of, 311-313
  - requirements, 311
- icons example, 310
- organization schemes, 317
- performance issues, 317
- requesting, 316
- runtime changes, handling, 318
- screen orientations, 316
- website, 321

**Amazon Appstore**, 393

**analog clocks**, 197

**AnalogClock control**, 197

**anddev.org forum**, 7

**Android**

- advantages, 22
- command-line tool, 98
- core files, 65

Debug Bridge (ADB), 87, 100

Developer resources

- blog, 350
- website, 6, 35

documentation, 83-85

- maintenance/support, 345
- mobile development projects, 337-338

SDK, 43

sections, 83

website, 83

licensing, 25

manifest file, 119

mascot, 24

overview, 5

project, 20

Project Wizard, 62-64

- application information, 64
- build targets, 63
- file locations, choosing, 63
- launch activities, 64
- minimum SDK version, 64
- names, 62
- package names, 64

Support (Compatibility) Package, 247

Tools Project Site website, 7

Virtual Devices (AVDs), 56

- creating, 65, 403-404
- display options, 404-405
- emulator, 401, 410
- hardware configuration settings, 405-407
- management website, 421
- Manager, 47-48, 402
- Snake application, creating, 56-57

Virtual Device Manager (AVD Manager), 47-48, 402

- Android Market, 385, 392**
  - alternatives, 393–394
  - applications
    - removing, 393
    - upgrading, 392
    - uploading, 387
  - contact/consent options, 390
  - developer accounts, creating, 385–387
  - fees, 390
  - filters, 381, 395
  - help, 390
  - listing details, 388
  - marketing assets, uploading, 388
  - packaging requirements, 381
  - publishing options, 390
  - return policies, 392
  - reviews, 344
  - website, 7, 385, 395
- android package, 44**
- AndroidManifest.xml file, 65**
- <animation-list> tag, 157**
- AnimationDrawable resources, 157**
- animations**
  - frame-by-frame, 156–157
  - storing, 139
  - tweened, 156
    - accessing, 158
    - defining, 157
    - overview, 157
    - spinning graphic example, 158
  - types, 156
- anonymous diagnostics, gathering, 354**
- APIs (application programming interfaces), 32**
  - ApiDemos sample app, 51
  - C2DM, 45
- Google, 45–46**
- legacy support, 45**
- levels website, 136**
- SDK versions, 127**
- third-party, 45**
- XML SAX support, 45**
- App Widgets, 172**
- application framework**
  - important packages, 43
  - third-party APIs, 45
- application programming interfaces.**
  - See APIs*
- Application tab (Eclipse), 121**
- applications**
  - Android Market, managing
    - removing, 393
    - return policies, 392
    - reviews, 344
    - upgrades, 392
  - assets, retrieving, 105
  - availability, 22
  - creating
    - AVD, 65
    - core files, 65
    - device target selection mode, 66
    - launch configuration, 66–67
    - new project, creating, 62–64
  - data
    - directories, 278–280
    - storing, 275–276
  - debugging
    - ADB, 87
    - emulator, 69–73
    - DDMS, 87
    - devices, 78–79

- designing
  - best practices, 355, 361
  - billing/revenue generation, 351–352
  - code diagnostics, 353, 358–359
  - code reviews, 358
  - coding standards, 357
  - feasibility, testing, 356
  - mistakes, avoiding, 355, 360
  - rules, 347–348
  - security, 351
  - single device bugs, 359
  - software process, choosing, 356
  - stability/responsiveness, 349–350
  - third-party standards, 352
  - tools, 354, 360
  - updates/upgrades, 354
  - user demands, meeting, 348
  - user interfaces, 348–349
  - development advantages, 26
  - feasibility assessments, 336
  - Flash support, 33
  - fragment-based, designing, 238
  - Google involvement, 22
  - Home screen, adding, 419
  - icons
    - configuring, 125
    - setting, 380
  - in-application billing, 379
  - installations, testing, 372
  - integration, 26
  - interoperability, 342
  - legacy support, 247–248
  - lifecycle, 34
  - location-based services, 77–78
- manifest file settings, 121
- names
  - configuring, 125
  - setting, 380
- native versus third-party, 33
- navigation, 115–116
- network-driven, 340
- operating system/hardware interaction, 34
- PeakBagger 1.0, 86
- performance, testing, 373
- popular, creating, 374
- preferences
  - accessing, 105, 267–268
  - adding, 264
  - appropriateness, 263
  - data types supported, 264
  - editing, 266–267
  - overview, 263
  - private, 264
  - reacting, 267
  - reading, 265
  - searching, 265
  - shared, 265
- release versions, testing, 384
- resources, retrieving, 105
- rs:ResEnum, 168
- running
  - MyFirstAndroidApp, 67–68
  - as operating system users, 31
- sample, 50–51
- signing, 382–384
- Snake
  - adding to Eclipse, 54
  - AVD, creating, 56–57

launch configuration, creating, 58-59  
missing folder error, 56  
running in emulator, 59-60  
sample code website, 81  
standalone, 340  
statistics collection, 379  
traceview, 375  
upgrades, testing, 371  
uploading to Android Market, 387  
usability, testing, 370  
versioning, 125  
visual appeal, 370  
web, 33

**ApplicationTestCase class, 359**

**apply( ) method, 267**

**architecture, 29**

- Dalvik VM, 31
- Linux operating system, 30

**ArrayAdapter class, 222-223**

**arrays**

- adapters, 222-223
- integers, 151
  - accessing, 150
  - creating, 151
  - defining, 150
  - storing, 139
- mixed-type, storing, 139
- strings, 149
  - accessing, 147-148
  - bold/italic/underline, 147
  - formatting, 146-147
  - overview, 146
  - plural, 149
  - storing, 139
- <string> tag, 146

**aspect ratio (screens), 313**

**assets**

- folder, 65
- retrieving, 105

**attributes**

- AutoCompleteTextView control, 181
- DatePicker control, 190
- debuggable, 382
- EditText control, 177
- FrameLayout class, 207-209
- GridLayout class, 216
- layout, 205-206
- LinearLayout class, 210
- Preference class, 269
- RelativeLayout class, 211
- TableLayout class, 214
- TextView control, 173-174
  - autoLink, 174-176
  - ellipsize, 174
  - ems, 174
  - lines, 174

**audio playback/recording support, 406**

**AutoCompleteTextView class, 179-181**

**autocompletion, 179-181. 443**

**autoLink attribute, 174-176**

**automating testing, 368**

**AVDs (Android Virtual Devices), 56**

- creating, 65, 403-404
- display options, 404-405
- emulator
  - launching, 411
  - settings, 401
- hardware configuration settings, 405-407
- management website, 421
- Manager, 47-48, 402
- Snake application, creating, 56-57

**B**

- 
- <b> tag, 147**
  - background layout example, 162-163**
  - backup services, testing, 373**
  - battery support, 406**
  - below attribute, 213**
  - best practices**
    - applications
      - popularity, increasing, 374
      - upgrades, 371
      - usability, 370
    - automation, 368
    - backup services, 373
    - billing, 373
    - black-box, 369
    - build validations, 368
    - conformance, 372
    - coverage, maximizing, 367
    - defect tracking systems, 363-365
    - development
      - code diagnostics, 358-359
      - code reviews, 358
      - coding standards, 357
      - feasibility, testing, 356
      - mistakes, avoiding, 360
      - reference websites, 361
      - single device bugs, 359
      - software process, choosing, 356
      - tools, leveraging, 360
    - device upgrades, 372
    - emulator, 399-401
      - benefits, 368
      - disadvantages, 369
    - environment, 365-367
    - installations, 372
  - integration points, 371**
  - internationalization, 372**
  - mistakes, 375**
  - performance, 373**
  - preproduction devices, 368**
  - reference websites, 376**
  - remote servers/services, 370**
  - third-party standards, 371**
  - tools, 374-375**
  - unexpected configuration changes, 373**
  - visual appeal, 370**
  - white-box, 369**
  - billing, 351-352**
    - in-app, 379
    - testing, 373
  - black-box testing, 369**
  - Blog section (documentation), 84**
  - bmgr tool, 98, 375**
  - bold, strings, 147**
  - book website, 6**
  - Booleans, 149**
    - accessing, 150
    - defining, 150
    - storing, 139
  - broadcast receivers, registering, 133**
  - broadcasting intents, 117**
  - Browser content provider, 285, 289-290, 300**
  - browsers**
    - early WAP, 16
    - information, accessing, 289-290
    - querying, 290
  - browsing files, 432**
  - build errors, 447**
  - build targets, choosing, 63**
  - build validations, 368**

**built-in content providers**, 285  
**bundles** activity state, saving, 110  
**Button class**, 183-184  
**buttons**, 183  
  basic, 183-184  
  check boxes, 183, 186  
  image buttons, 185  
  radio, 183, 187-189  
  switches, 183, 187  
  toggle, 183, 186

## C

---

**C2DM (Cloud to Device Messaging)**, 45  
**cache files**, storing, 282-283  
**cache subdirectory**, retrieving, 279  
**CalendarContract content provider**, 285, 291  
**calendars**, accessing, 291  
**calendarViewShown attribute**, 190  
**callbacks**  
  activities, 107-109  
    initializing/retrieving activities, 109  
    static data, initializing, 108  
    stopping/saving/releasing, 109  
  fragments, 237  
**calling**  
  between emulators, 413  
  logs, 288-289  
**CallLog content provider**, 285, 288-289, 300  
**camera support**, 406  
**centerHorizontal attribute**, 212  
**centerInParent attribute**, 212  
**centerVertical attribute**, 212  
**certificate signing**, 32  
**character picker dialogs**, 252  
**CharacterPickerDialog class**, 252, 260

**check boxes**, 183, 186  
**CheckBox class**, 183, 186  
**CheckBoxPreference class**, 274  
**choosing**  
  build targets, 63  
  devices for tracking, 332  
  distribution methods, 377-378  
  file locations, 63  
  primary entry points, 132  
  software processes, 356  
    iterative, 327  
    waterfall approaches, 326  
  source control systems, 339  
  versioning schemes, 339-340  
**Chronometer control**, 195-196  
**circular progress indicators**, 192  
**classes**  
  AbsoluteLayout, 214  
  Activity  
    callbacks, 107-109  
    defined, 103  
    website, 81, 118  
  AdapterView, 222-224  
  ActivityUnitTestCase, 359  
  AlertDialog, 252, 260  
  AnalogClock, 197  
  AnimationDrawable, 157  
  App Widgets, compared, 172  
  ApplicationTestCase, 359  
  ArrayAdapter, 222-223  
  AutoCompleteTextView, 179-181  
  Button, 183-184  
  CharacterPickerDialog, 252, 260  
  CheckBox, 183, 186  
  CheckBoxPreference, 274

- Chronometer, 195-196
- ContactsContract, 296
- containers, 221
- Context
  - defined, 103
  - methods, 278-279
  - permission modes, 277
  - website, 118, 284
- creating, 443
- CursorAdapter, 222-224
- DatePicker, 190-191
- DatePickerDialog, 253, 260
- Debug, 373
- Dialog, 252, 260, 321
- DialogFragment, 238, 250, 260
- DigitalClock, 196
- DisplayMetrics, 304
- EditText, 176-179
- EditTextPreference, 274
- Environment, 284
- FieldNoteListFragment, 240-243
- FieldNoteViewActivity, 246
- FieldNoteWebViewFragment, 243-244
- File, 282-284
- Fragment
  - callback methods, 237
  - defined, 104
  - reference website, 250
  - subclasses, 237
  - website, 118
- FrameLayout, 207
  - attributes, 207-209
  - website, 231
- Gallery, 231
- GalleryView, 222
- GridLayout, 216-220, 231
- GridView, 222, 231
- ImageButton, 185
- imports (Eclipse), 73
- Intent, 104
- LayoutParams, 205
- LinearLayout, 209-210, 231
- ListActivity, 225, 231
- ListFragment, 238
  - implementing, 240-243
  - reference website, 250
- ListPreference, 274
- ListView, 222, 231
- LocationManager, 81
- Log
  - methods, 73
  - website, 81
- MarginLayoutParams, 205
- MediaPlayer
  - methods, 74
  - website, 81
- MediaStore, 286
- MoreAsserts, 359
- MultiAutoCompleteTextView, 181
- OnItemClickListener, 224
- PerformanceTestCase, 359
- Preference, 269, 274
- PreferenceActivity, 268-273
- PreferenceCategory, 274
- PreferenceFragment, 238, 250
- PreferenceScreen, 274
- ProgressBar, 192-193
- ProgressDialog, 253, 260
- RadioButton, 183, 187-189

RatingBar, 194-195  
RelativeLayout, 211-214, 231  
SeekBar, 194  
Service, 104, 116  
ServiceTestCase, 359  
Settings, 292  
ShapeDrawable, 153  
SharedPreferences, 105, 264  
SimpleFragDialogActivity, 258  
SlidingDrawer, 230  
Spinner, 181-183  
StrictMode, 373  
Switch, 183, 187  
TabActivity, 226-228, 231, 238  
TabHost, 226, 229-231  
TableLayout, 214-216, 231  
TabSpecs, 226  
TextView, 173  
TimePicker, 191  
TimePickerDialog, 253, 260  
ToggleButton, 183, 186  
TouchUtils, 359  
Uri  
    methods, 74  
    website, 81  
View, 171, 204  
ViewAsserts, 359  
ViewGroup, 204  
    layout attributes, 205-206  
    layout subclasses, 204  
    View containers, 204  
    View class, compared, 204  
    website, 231  
WebView, 243-244  
WebViewFragment, 238, 250

**clean states (devices), 366-367**

**clear( ) method, 266**

**clearCheck( ) method, 189**

**click events, handling, 224**

**clients, testing, 337**

**clocks**

- analog, 197
- digital, 196

**cloud, testing, 337**

**Cloud to Device Messaging (C2DM), 45**

**code**

- autocomplete, 443
- build errors, 447
- classes, creating, 443
- comments, 446
- diagnostics, 358-359
- formatting, 444
- imports, 443
- methods, creating, 443
- packaging preparations, 380
  - Android Market, 381
  - application names/icons, 380
  - debugging/logging, turning off, 381
  - manifest file, 381
  - permissions, 382
  - target platforms, 381
  - versioning, 380
- QuickFix feature, 446
- refactoring, 444-445
- reorganizing, 446
- reviews, 358
- standards, defining, 357
- stepping through (Eclipse), 72
- versioning, 380

**collapseColumns attribute, 214**

**<color> tag, 151**

**colors, 151**

- accessing, 151
- defining, 151
- formats, 151
- storing, 139

**column attribute**

- GridLayout class, 218
- TableLayout class, 215

**columnCount attribute, 217**

**columnSpan attribute, 218**

**command-line tools, 98**

- bmgr, 98, 375
- etc1tool, 99
- Exerciser Monkey, 373, 376
- layoutopt, 220
- logcat, 99, 374, 436–437
- mksdcard, 99
- sqlite3, 99, 375
- zipalign, 99

**commands**

- File menu
  - New, Android Project, 62
  - New, Other, 54
- Run menu
  - Debug Configurations, 58, 66
  - Run Configurations, 66

**comments (code), 446**

**commercializing WAP, 16**

**commit( ) method, 266**

“Common Tasks and How to Do Them in Android” website, 81

**commonly used packages, 34**

**compatibility, 301**

- alternative resources
  - benefits, 308
  - creating, 309

**data, saving, 318**

**defined, 308**

**directory qualifiers, 309–313**

**icons example, 310**

**organization schemes, 317**

**performance issues, 317**

**requesting, 316**

**runtime changes, handling, 318**

**screen orientations, 316**

**best practices website, 322**

**densities, 301**

**fragments, 305**

**Google TVs, 319–321**

**maximizing, 303**

**nine-patch stretchable graphics, 306**

**OpenGL versions, 301**

**platforms, 301**

**runtime changes, handling, 321**

**screen sizes, 301**

**types, 305–306**

**Support (Compatibility) Package, 118, 248–249, 305**

**tablets, 318–319**

**user interfaces, 303–304**

**working squares, 306–307**

**<compatible-screens> tag, 131**

**competitive hardware, 23**

**complete platform, 23**

**completionHint attribute, 181**

**completionThreshold attribute, 181**

**Conder, Shane**

- blog, 35
- contacting, 8

**configuring**

- alternative resource directory qualifiers, 311
- development environment
  - installation, 38
  - software requirements, 37–38
- dialogs, 256
- emulator location, 77
- GL texture compression formats, 131
- hardware for debugging, 39–41
- icons, 125
- launches, 66–67
- operating systems for device
  - debugging, 39
  - screen sizes, 130

**conformance, testing, 372****consent options (Android Market), 390****console (emulator)**

- commands, 419
- connecting, 416
- GPS coordinates, sending, 418
- incoming calls, simulating, 416
- network status, displaying, 418
- power settings, 418
- SMS messages, simulating, 416

**contact options (Android Market), 390****contacting Conder/Darcey, 8****contacts**

- accessing, 292
- adding, 297–298
- content provider, 292–294
  - permissions, 292
  - querying quickly, 294–295
  - records, 297–299
  - website, 300

deleting, 298–299

legacy support, 293–294

permissions, 292

querying

Contacts provider, 293–294

ContactsContract content provider, 295

quickly, 294–295

updating, 298

**ContactsContract content provider, 286, 292**

data column classes, 296

new features, 295

overview, 295

permissions, 292

queries, 295

querying quickly, 294–295

website, 297, 300

**containers (view), 204, 221**

data-driven, 221–222

adapters, 222–225

arrays, 222–223

click events, handling, 224

data, binding, 224

databases, 223–224

lists of items, 225

scrolling, 229

sliding drawers, 230

switchers, 230

tabs, 226–228

**contains( ) method, 265****content providers, 285**

accessing, 286

AlarmClock, 285

Browser, 285, 289–290

built-in, 285

- CalendarContract, 285, 291
- CallLog, 285, 288–289
- contacts, 292
  - legacy Contacts provider, 293–294
  - permissions, 292
  - querying quickly, 294–295
- ContactsContract, 286
  - data column classes, 296
  - new features, 295
  - overview, 295
  - queries, 295
  - website, 297
- MediaStore, 286–288
  - audio file titles, retrieving, 287
  - classes, 286
  - permissions, 289
  - records
    - adding, 297–298
    - deleting, 298–299
    - updating, 298
  - reference websites, 300
- registering, 133
- SearchRecentSuggestions, 286
- Settings, 286, 292
- testing, 285
- third-party, 299
- UserDictionary, 286, 291
- VoiceMailContract, 286, 291
- context**
  - assets, retrieving, 105
  - defined, 103
  - files/directories, 105
  - preferences, 105
  - resources, 105
  - retrieving, 104
- Context class**
  - defined, 103
  - methods, 278–279
  - permission modes, 277
  - website, 118, 284
- contextual links, creating, 174–176**
- controls, 171. See also classes**
  - layout, 172
  - scrolling, 229
  - switchers, 230
    - attributes, 173–174
    - contextual links, 174–176
    - height, 174
    - width, 174
- copying files, 432–433**
- core files, 65**
- costs, 23**
- country codes website, 322**
- coverage, maximizing, 367**
  - application
    - popularity, increasing, 374
    - upgrades, 371
    - usability, 370
  - automation, 368
  - backup services, 373
  - billing, 373
  - black-box testing, 369
  - build validations, 368
  - conformance, 372
  - device upgrades, 372
  - emulators
    - benefits, 368
    - disadvantages, 369
  - installations, 372
  - integration points, 371

internationalization, 372  
performance, 373  
preproduction devices, 368  
remote servers/services, 370  
third-party standards, 371  
unexpected configuration changes, 373  
visual appeal, 370  
white-box testing, 369

**crashes, monitoring**, 345

**create( ) method**, 75

**createTabContent( ) method**, 228

**creating**

- AVDs, 65
- intents, 114
- launch
  - activities, 64
  - configurations, 66–67
- new projects, 62–64
  - application information, 64
  - build targets, 63
  - file locations, choosing, 63
  - launch activities, 64
  - minimum SDK version, 64
  - names, 62
  - package names, 64
- Nine-Patch Stretchable Graphics, 95–97
- resources, 143–146

**CursorAdapter class**, 222-224

**cursors**, 223-224

**customization development method**, 328-329

**customizing**. *See also* **configuring**

- alternative resource directory qualifiers, 311

development environment  
installation, 38  
software requirements, 37–38  
dialogs, 256  
emulator location, 77  
GL texture compression formats, 131  
icons, 125  
launches, 66–67  
screen sizes, 130

## D

---

**D-Pad support**, 406

**Dalvik Debug Monitor Server**. *See* **DDMS**

**dalvik package**, 44

**Dalvik VM**, 31

**dangerous protection level (permissions)**, 135

**Darcey, Lauren**

- blog, 35
- contacting, 8

**data**

- alternative resources, saving, 318
- applications
  - directories, 278
  - storing, 275–276
- devices, storing, 332
- intents, 114
- preference values, 264

**data-driven containers**, 221-222

- adapters, 222–225
- arrays, 222–223
- click events, handling, 224
- data, binding, 224
- databases, 223–224
- lists of items, 225

**data/app directory**, 432

**data/data/<package name>/ cache/ directory, 432**

**data/data/<package name>/ databases/ directory, 432**

**data/data/<package name>/ directory, 432**

**data/data/<package name>/ files/ directory, 432**

**databases, 330**

- adapters, 223–224
- data, storing, 332
- devices for tracking, choosing, 332
- functions, 332
- third-party, 333

**DatePicker control, 190–191**

**DatePickerDialog class, 253, 260**

**dates**

- dialogs, 253
- picker dialogs, 253
- user input, 190–191

**DDMS (Dalvik Debug Monitor Server), 47, 87, 423**

- debuggers, attaching, 425
- debugging applications, 87
- drag-and-drop operations, 433
- Emulator Control pane, 434
- features, 424
- File Explorer, 430
- browsing, 432
- copying files, 432–433
- deleting files, 433
- drag-and-drop operations, 433
- garbage collection, 428
- heap statistics, monitoring, 427
- HPROF files, 429
- logging, 436–437
- memory allocations, 430
- perspective (Eclipse), 47

processes, stopping, 426

screen captures, 435

standalone application, 423

threads, monitoring, 426

website, 100

**Debug class, 373**

**Debug Configurations command (Run menu), 58, 66**

**debuggable attribute, 382**

**debugging. See also troubleshooting**

- ADB, 87
- configurations, 66, 409
- devices, 78–79
- Eclipse
- perspective, 47
- stepping through code, 72
- emulator, 69–73
- hardware, enabling, 39–41
- MyFirstAndroidApp, 66
- operating systems, configuring, 39
- turning off, 381
- user interfaces
- drawing issues, 92–93
- layout control organization, 94
- websites, 100

**default resources, 141–142, 308**

**defaultValue attribute, 269**

**defect tracking systems, 363**

- information, logging, 363
- types of defects, 364–365

**delete( ) method, 299**

**deleteFile( ) method, 278**

**deleting**

- activity static data, 110
- applications from Android Market, 393
- contacts, 298–299

dialogs, 255–256  
files, 278, 433  
preferences, 266–267

**densities, compatibility**, 301

**deprecated methods**, 252

**designing applications**

- best practices, 355
- code diagnostics, 358–359
- code reviews, 358
- coding standards, 357
- feasibility, testing, 356
- mistakes, avoiding, 360
- reference websites, 361
- single device bugs, 359
- software process, choosing, 356
- tools, leveraging, 360

billing/revenue generation, 351–352

- in-app, 379
- testing, 373

diagnostics, leveraging, 353

mistakes, avoiding, 355

rules, 347–348

security, 351

stability/responsiveness, 349–350

third-party standards, 352

tools, 354

updates/upgrades, 354

user demands, meeting, 348

user interfaces, 348–349

**destroying.** See **deleting**

**Dev Guide section (documentation)**, 84

**developers**

- accounts, creating, 385–387
- registration, 32

**Developer.com website**, 7

**“Developing on a Device” website**, 81

**development environment**

- configuring
- hardware, 39–41
- installation, 38
- operating systems, 39
- software requirements, 37–38

SDK, upgrading, 41

testing, 365

- clean states, 366–367
- device configurations, 365–366
- real world simulations, 367

testing with Snake application, 53

- adding to Eclipse, 54
- AVD, creating, 56–57
- launch configuration, creating, 58–59
- running in emulator, 59–60

**development tools**, 25

- ADB, 87
- Android command-line tool, 98
- Android documentation, 83–85
- AVD Manager, 47–48
- application design, 354
- bmgr, 98, 375
- DDMS, 47, 87
  - debuggers, attaching, 425
  - debugging applications, 87
  - drag-and-drop operations, 433
  - Emulator Control pane, 434
  - features, 424
  - File Explorer, 430–433
  - garbage collection, 428
  - heap statistics, monitoring, 427
  - HPROF files, 429
  - logging, 436–437
  - memory allocations, 430

- perspective (Eclipse), 47
- processes, stopping, 426
- screen captures, 435
- standalone application, 423
- threads, monitoring, 426
- website, 100
- development, leveraging, 360
- dmtracedump, 98
- Eclipse ADT plug-in, 46–47
  - creating resources, 143–146
  - editors, 89
  - layout resource editor, 164–166
  - UI designer, 90
- emulator, 49
  - AVDs, 401–407
  - benefits, 368
  - best practices, 399–401
  - calling between, 413
  - console, 416–419
  - Control pane, 434
  - disadvantages, 369
  - files, managing, 432–433
  - GPS location, 411–413
  - Hierarchy Viewer, launching, 92
  - icon tips, 419
  - launching, 60, 407–411
  - limitations, 420–421
  - location, configuring, 77
  - messaging between, 415
  - MyFirstAndroidApp, 67–73
  - overview, 85, 399
  - pages, adding, 419
  - PeakBagger 1.0, 86
  - performance, 407–408
  - screen captures, 435
- smartphone-style example, 49
- Snake app, running, 59–60
- startup options, 408
- tablet-style example, 50
- tips, 419
- unlocking, 60
- wallpaper, editing, 419
- website, 86, 100, 421
- etc1tool, 99
- Exerciser Monkey, 373, 376
- Extract Local Variable, 445
- Extract Method, 445
- Hierarchy Viewer
  - launching, 92
  - layout controls above application content, 92
  - Layout View mode, 92–93
  - modes, 92
  - online tutorial, 94
  - overview, 91
  - Pixel Perfect mode, 94
  - user interface optimization, 94
- hprof-conv, 98
- layoutopt, 220, 375
- logcat, 86–87, 99, 374, 436–437
- mksdcard, 99
- monkey, 99
- monkeyrunner, 99
- Nine-Patch Stretchable Graphics, 95–97
- Rename, 444
- resource editors, 89
- sample applications, 50–51
- SDK Manager, 47–48
- signing, 384

- sqlite3, 99
  - testing, 374–375
  - traceview, 98
  - UI designer, 90
  - website, 98
  - zipalign, 99
- devices**
- application interaction, 34
  - availability, 334–335
  - bugs, handling, 359
  - clean states, 366–367
  - compatibility, 301
    - alternative resources. *See* compatibility, alternative resources
  - best practices, 322
  - densities, 301
  - fragments, 305
  - Google TVs, 319–321
  - maximizing, 303
  - nine-patch stretchable graphics, 306
  - OpenGL versions, 301
  - platforms, 301
  - runtime changes, handling, 321
  - screen sizes, 301
  - screen types, 305–306
  - Support (Compatibility) Package, 305
  - tablets, 318–319
  - user interfaces, 303–304
  - working squares, 306–307
- competitive, 23
  - convergence, 17
  - databases, 330
    - data, storing, 332
  - devices for tracking, choosing, 332
  - functions, 332
  - third-party, 333
- debugging
    - configuring, 39–41
    - enabling, 39–41
  - MyFirstAndroidApp, 78–79
  - operating systems, configuring, 39
- files
- browsing, 432
  - copying, 432–433
  - deleting, 433
  - storing, 332
- fragmentation, 365–366
- Google Experience, 335
- history
- Android manufacturers, 20
  - Android project, 20
  - device convergence, 17
  - first Android phone, 20
  - first cell phone, 13
  - first-generation, 13
  - form factors, 15
  - functionality, 11–13
  - Google Internet model, 19
  - Open Handset Alliance (OHA), 20
  - operators, 21
  - platform market penetration, 18–19
  - proprietary platforms, 17
  - time-waster games, 14
  - WAP, 15–16
- limitations, 340
- loss of signal, 367
- personal, testing, 330
- preproduction, 368
- profiles, 405–407
- RAM size, 405
- required features, specifying, 129–130

screen  
 captures, 435  
 sizes, configuring, 130, 301  
 types, 305–306

settings, accessing, 292

target  
 acquiring, 335  
 availability, 334–335  
 identifying, 333  
 selection mode, 66

testing, 336

third-party firmware  
 modifications, 365

tracking, 332

unexpected configuration changes,  
 testing, 373

upgrades, 372

USB connections, 409

**diagnostics**  
 anonymous, gathering, 354  
 code, 358–359  
 leveraging, 353

**Dialog class, 252**  
 reference website, 260  
 website, 321

**DialogFragment class, 238, 250, 260**

**dialogs**  
 adding  
 activities, 254  
 fragment-based method, 252  
 legacy method, 251

alerts, 252

basic, 252

character pickers, 252

date pickers, 253

fragment method, 257–260

legacy method, 253  
 customizing, 256  
 defining, 254  
 destroying, 255–256  
 dismissing, 255  
 initializing, 254  
 launching, 255  
 lifecycle, 254  
 progress, 253  
 time pickers, 253  
 types, 252–253  
 websites, 260

**dictionary, 291**

**digital clocks, 196**

**DigitalClock control, 196**

**<dimen> tag, 152**

**dimensions, 152**  
 accessing, 153  
 defining, 152  
 resource file example, 152  
 screens, 312  
 storing, 139  
 unit measurements, 152

**directories**  
 accessing, 105  
 applications, 278  
 cache files, 282–283  
 default application  
 reading files, 280  
 writing files, 279  
 files, listing, 278, 282  
 important, 432  
 qualifiers, 309–313  
 resources, 138  
 subdirectories  
 creating, 282  
 retrieving, 279

**disadvantages**, 28

**dismissDialog( ) method**, 254

**dismissing dialogs**, 255

**displaying**

- AVDs, 404-405
- data to users
  - adjusting progress, 194
  - clocks, 196-197
  - progress bars, 192-193
  - ratings, 194-195
  - time passage, 195-196
- Eclipse windows side by side, 440
- network status, 418
- TextView control, 173-174

**DisplayMetrics class**, 304

**distribution**, 25, 343-344, 385. *See also publication*

- ad revenue, 379
- alternatives, 393-394
- Android Market, 385, 392
  - contact/consent options, 390
  - developer accounts, creating, 385-387
  - fees, 390
  - help, 390
  - listing details, 388
  - marketing assets, uploading, 388
  - publishing options, 390
  - removing applications, 393
  - return policies, 392
  - upgrading applications, 392
  - uploading applications, 387
  - website, 385
- choosing, 377-378
- in-application billing, 379
- intellectual property protection, 378

- market reviews, 344
- options, 27-28
- self-distribution, 394-395
- statistics collection, 379

**dmtracedump tool**, 98

**dock mode**, 313

**Document Object Model Core package**, 45

**documentation**, 83-85

- maintenance/support, 345
- mobile development projects, 337-338
- SDK, 43
- sections, 83
- website, 83

**<drawable> tag**, 153-154

**drawables**

- accessing, 154
- defining, 153
- ShapeDrawable class, 153
- storing, 139

## E

---

**Eclipse**, 38

- ADT plug-in, 46-47, 88
- creating resources, 143-146
- resource editors, 89
- layout resource editor, 164-166
- UI designer, 90
- applications, signing, 383
- AVDs, creating, 403-404
- code
  - autocomplete, 443
  - build errors, 447
  - classes, creating, 443
  - comments, 446
  - formatting, 444

imports, 443  
 methods, creating, 443  
 QuickFix feature, 446  
 refactoring, 444–445  
 reorganizing, 446  
 DDMS perspective, 87  
 debug configurations, creating, 409  
 device USB connections, 409  
 emulator GPS location, creating, 412  
 imported classes, 73  
 layouts, designing, 164–166  
 log filters, creating, 441  
 manifest files  
     application-wide settings, 121  
     editing, 120  
     instrumentation settings, 122  
     package-wide settings, 121  
     permissions, 121  
 perspectives, 47, 440  
 projects, creating, 62–64  
     application information, 64  
     build targets, 63  
     file locations, choosing, 63  
     launch activities, 64  
     minimum SDK version, 64  
     names, 62  
     package names, 64  
 Rename tool, 444  
 resources, creating, 143–146  
 run configurations, creating, 409  
 sample projects, adding, 52  
 searches, 442  
 Snake application  
     adding, 54  
 AVD, creating, 56–57  
 launch configuration, creating, 58–59  
 running, 59–60  
 source control services, 439  
 stepping through code, 72  
 tabs, closing, 441  
 tasks, 442  
 two sections of same file, viewing, 441  
 website, 52  
 windows  
     maximizing, 440  
     minimizing, 440  
     open, limiting, 441  
     viewing side by side, 440

**edit( ) method, 265**

**editing**

- content providers, records
  - adding, 297–298
  - deleting, 298–299
  - updating, 298
- manifest file, 120
  - application-wide settings, 121
  - Eclipse, 120
  - instrumentation settings, 122
  - manually, 123–124
  - package-wide settings, 121
  - permissions, 121
  - preferences, 266–267
  - wallpaper, 419

**EditText class, 176–178**

- defining, 177
- hints, 177
- InputFilter interface, 178–179
- lines, 177

**EditTextPreference class, 274**

**elapsedRealtime( ) method, 196**

**ellipsis (...), 174**

**ellipsize attribute, 174**

**ems, 174**

**emulator, 49**

AVDs

    AVD Manager, 402

    creating, 403–404

    display options, 404–405

    hardware configuration settings, 405–407

    settings, 401

benefits, 368

best practices, 399–401

calling between, 413

console

    commands, 419

    connecting, 416

    GPS coordinates, sending, 418

    incoming calls, simulating, 416

    network status, displaying, 418

    power settings, 418

    SMS messages, simulating, 416

Control pane, 434

disadvantages, 369

files, managing, 432–433

GPS location, 411–413

Hierarchy Viewer, launching, 92

icon tips, 419

launching, 60

    AVD Manager, 411

    options, 407

    specific projects, 408–409

limitations, 420–421

location, configuring, 77

messaging between, 415

**MyFirstAndroidApp**

    debugging, 69–73

    running, 67–68

overview, 85, 399

pages, adding, 419

PeakBagger 1.0, 86

performance, 407–408

screen captures, 435

smartphone-style example, 49

Snake app, running, 59–60

startup options, 408

tablet-style example, 50

tips, 419

unlocking, 60

wallpaper, editing, 419

website, 86, 100, 421

**End User License Agreement (EULA), 351**

**enforcing**

permissions, 134

platform requirements, 129–130

    device features, 129–130

    input methods, 129

    screen sizes, 130

    system requirements, 125–128

**entries attribute, 183**

**environment**

configuring

    hardware, 39–41

    installation, 38

    operating systems, 39

    software requirements, 37–38

SDK, upgrading, 41

testing, 365  
clean states, 366–367  
device configurations, 365–366  
real world simulations, 367  
testing with Snake application, 53  
adding to Eclipse, 54  
AVD, creating, 56–57  
launch configuration, creating,  
58–59  
running in emulator, 59–60

**Environment class, 284**

**etc1tool tool, 99**

**EULA (End User License Agreement), 351**

**Exerciser Monkey tool, 373, 376**

**expansion beyond smartphones, 23**

**extensibility, 341–342**

**external activities, launching, 114**

**external file storage, 283–284**

**external libraries, 130**

**Extract Local Variable tool, 445**

**Extract Method tool, 445**

**extreme programming website, 346**

---

## F

### feasibility

assessments, 336  
testing, 356

### FieldNoteListFragment class, 240–243

### FieldNoteViewActivity class, 246

### FieldNoteWebViewFragment class, 243–244

### FierceDeveloper newsletter website, 7

### File class, 282–284

### File Explorer, 430

browsing, 432  
copying files, 432–433

deleting files, 433  
drag-and-drop operations, 433

### File menu commands

New, Android Project, 62  
New, Other, 54

### fileList( ) method, 278

### files. *See also* directories

accessing, 105, 278  
AndroidManifest.xml, 65  
browsing, 432  
cache  
storing, 282–283  
subdirectory, retrieving, 279  
copying, 432–433  
core, 65  
creating, 282  
default application directories,  
279–280  
deleting, 278, 433  
files subdirectory, 279  
gen/com/androidbook/  
myfirstandroidapp/R.java, 65  
handles, retrieving, 278  
HPROF, creating, 429  
important directories, 432  
listing, 278, 282  
managing  
best practices, 276–277  
methods, 278–279

### manifest

activities, registering, 131–133  
applications, 125, 134  
broadcast receivers, registering, 133  
content providers, registering, 133  
editing, 89, 120–124

external libraries, 130  
features, 135  
GL texture compression formats, 131  
names, 119  
overview, 119-120  
package names, 124  
permissions, 133-135  
platform requirements, enforcing, 129-130  
publication preparations, 381  
reference website, 136  
screen compatibility, 131  
SDK versions, targeting, 126-128  
services, registering, 133  
system requirements, enforcing, 125-128  
opening, 279  
permissions, 277  
private, 277  
`proguard.cfg`, 65  
`project.properties`, 65  
raw, 160  
    accessing, 161  
    defining, 160  
    reading byte-by-byte, 280  
    storing, 140  
readable, 277  
reading  
    byte-by-byte, 280  
    XML files, 280-281  
`res/drawable`, 65  
`res/layout/main.xml`, 65  
`res/values/strings.xml`, 65  
resources  
    accessing, 142  
    aliases, 162  
    alternative, 141-142. *See also resources, alternative*  
    animations, 139, 156  
    application, retrieving, 105  
    Boolean, 139, 149-150  
    colors, 139, 151  
    creating, 143-146  
    default, 141-142, 308  
    defined, 137  
    dimensions, 139, 152-153  
    directories, 138  
    drawables, 139, 153-154  
    frame-by-frame animations, 156-157  
    images, 139-141, 154-155  
    integers, 139, 150-151  
    layouts. *See resources, layouts*  
    menus, 139, 158-159  
    mixed-type arrays, 139  
    primitive, 140  
    raw files, 140, 160-161  
    references, 161-162  
    selectors, 156  
    storing, 137-140  
    strings, 139, 146-149  
    styles, storing, 140  
    system, 167-168  
    themes, storing, 140  
    tweened animations, 157-158  
    types, 138-140  
    user preferences, 269-272  
    websites, 168  
    XML files, 139, 159-160

source control services, 439  
storing, 141, 283–284  
subdirectory, 279  
writeable, 277  
XML, 159  
    accessing, 160  
    defining, 160  
    parsing utilities, 280  
    reading, 280–281  
    storing, 139

**filter( ) method, 179**

**filters**  
    Android Market, 381  
    input, 178–179  
    intents, 114  
    logging, 441  
    market, 395

**firmware upgrades, testing, 345**

**first Android handset, 20**

**first cell phone, 13**

**first-generation mobile phones, 13**

**Flash support, 33**

**folders, 65**

**footers, 226**

**forceError( ) method, 69**

**Foreground attribute, 208**

**foregroundGravity attribute, 208**

**form factors (phones), 15**

**formats**  
    code, 444  
    colors, 151  
    images, 154  
    resource references, 161  
    strings, 146–147

**forms**

    autocomplete, 179–181  
    LinearLayoutview, 209–210

**forums, 7****Fragment class**

    callback methods, 237  
    defined, 104  
    reference website, 250  
    subclasses, 237  
    website, 118

**<fragment> tag, 235****fragments**

    Activity classes, creating, 246–247  
    applications, designing, 238  
    attaching, 237  
    compatibility, 305  
    creating, 237  
    defined, 104, 111  
    defining, 235  
    destroying, 237  
    detaching, 237  
    dialogs, 238, 257–260  
    layout resource files, creating, 244–246  
    legacy support, 247–248  
    lifecycle, 234–235  
    ListFragment, implementing, 240–243  
    lists, 238  
    MP3 music player example, 111–113  
    overview, 233  
    pausing, 237  
    resuming activity, 237  
    screen workflow, 234  
    specialty, 237  
    stopping activity, 237

updates, 236  
 user preferences, 238  
 visibility, 237  
 web content, 238  
 website, 118  
 WebView, implementing, 243–244  
 workflow flexibility, improving,  
 111–113  
**frame-by-frame animations**, 156–157  
**FrameLayout class**, 207  
 attributes, 207–209  
 website, 231  
**framework**  
 distribution, 25  
 important packages, 43  
 third-party APIs, 45  
**free development**, 23  
**free platform**, 24

**G**


---

**galleries**, 205, 222  
**Gallery class**, 231  
**GalleryView class**, 222  
**game application activities**, 106  
**garbage collection**, 428  
**gen/com/androidbook/myfirstandroidapp/**  
 R.java file, 65  
**getAll( ) method**, 265  
**getApplicationContext( ) method**, 104  
**getAssets( ) method**, 105  
**getBoolean( ) method**, 150, 265  
**getCacheDir( ) method**, 279, 282  
**getColor( ) method**, 151  
**getContentResolver( ) method**, 298  
**getDimension( ) method**, 153

**getDir( ) method**, 279  
**getExternalCacheDir( ) method**, 279,  
 283–284  
**getExternalFilesDir( ) method**, 279, 284  
**getExternalStoragePublicDirectory( )**  
 method, 284  
**getExternalStorageState( ) method**, 284  
**getFilesDir( ) method**, 279  
**getFileStreamPath( ) method**, 279  
**getFloat( ) method**, 265  
**getInt( ) method**, 265  
**getLocation( ) method**, 78  
**getLong( ) method**, 265  
**getResources( ) method**, 105  
**getSharedPreferences( ) method**, 105  
**getString( ) method**, 266  
**getStringSet( ) method**, 266  
**getText( ) method**, 173, 178  
**GL texture compression formats**,  
 configuring, 131  
**GNU General Public License Version 2**  
 (GPLv2), 25  
**Google**  
 Analytics, 45, 361  
 Android project, 20  
 APIs, 45  
 application development, 22  
 common intents, 115  
**Developers**  
 blog, 350  
 Guide website, 52, 100  
 Experience devices, 335  
 Internet model, 19  
 Maps, 413  
 Billing SDK, 46

- Market Licensing, 46
  - Open Handset Alliance (OHA), 20
    - manufacturers, 20
    - mobile operators, 21
    - website, 35
  - Team Android Apps website, 7
  - TV compatibility, 319–321
  - GPS support, 406**
    - coordinates
      - sending, 418
      - website, 413
    - emulator location, 77, 411
    - last known location, 77
    - MyFirstAndroidApp, 76–78
  - GPU emulation, 406**
  - graphics, 154**
    - accessing, 155
    - buttons, 185
    - drawables
      - accessing, 154
      - defining, 153
      - ShapeDrawable class, 153
    - formats, 154
    - GL texture compression formats, configuring, 131
    - Nine-Patch Stretchable Graphics, 95, 100
      - accessing, 155
      - creating, 95–97, 155
      - defined, 95, 155
      - device compatibility, 306
      - scaling, 95
      - storing, 139–141
    - screen captures, 435
  - gravity attribute**
    - GridLayout class, 218
    - LinearLayout class, 210
    - RelativeLayout class, 212
  - green robot, 24**
  - GridLayout class, 216–220, 231**
  - grids, 222**
  - GridView class, 222, 231**
  - groups**
    - permissions, 135
    - radio buttons, 187–189
  - growing platform, 28**
  - GSM modem support, 406**
- 
- H**
- Handango, 393**
  - handles (files), retrieving, 278**
  - handling**
    - click events, 224
    - runtime changes, 318, 321
    - single device bugs, 359
  - hardware**
    - application interaction, 34
    - availability, 334–335
    - bugs, handling, 359
    - clean states, 366–367
    - compatibility, 301
      - alternative resources. *See* compatibility, alternative resources
      - best practices, 322
      - densities, 301
      - fragments, 305
      - Google TVs, 319–321
      - maximizing, 303

- nine-patch stretchable graphics, 306
- OpenGL versions, 301
- platforms, 301
  - runtime changes, handling, 321
  - screen sizes, 301
  - screen types, 305–306
  - Support (Compatibility) Package, 305
  - tablets, 318–319
  - user interfaces, 303–304
  - working squares, 306–307
- competitive, 23
- convergence, 17
- databases, 330
  - data, storing, 332
  - devices for tracking, choosing, 332
  - functions, 332
  - third-party, 333
- debugging
  - configuring, 39–41
  - enabling, 39–41
- MyFirstAndroidApp, 78–79
- operating systems, configuring, 39
- files
  - browsing, 432
  - copying, 432–433
  - deleting, 433
  - storing, 332
- fragmentation, 365–366
- Google Experience, 335
- history
  - Android manufacturers, 20
  - Android project, 20
  - device convergence, 17
  - first Android phone, 20
- first cell phone, 13
- first-generation, 13
- form factors, 15
- functionality, 11–13
- Google Internet model, 19
- Open Handset Alliance (OHA), 20
- operators, 21
  - platform market penetration, 18–19
  - proprietary platforms, 17
  - time-waster games, 14
  - WAP, 15–16
- limitations, 340
- loss of signal, 367
- personal, testing, 330
- preproduction, 368
- profiles, 405–407
- RAM size, 405
- required features, specifying, 129–130
- screen
  - captures, 435
  - sizes, configuring, 130, 301
  - types, 305–306
- settings, accessing, 292
- target
  - acquiring, 335
  - availability, 334–335
  - identifying, 333
  - selection mode, 66
- testing, 336
- third-party firmware modifications, 365
- tracking, 332
  - unexpected configuration changes, testing, 373
- upgrades, 372
- USB connections, 409

- hash symbols (#), 151**
  - headers, 226**
  - heap statistics, monitoring, 427**
  - height**
    - input boxes, 177
    - text, 174
  - Hierarchy View perspective (Eclipse), 47**
  - Hierarchy Viewer**
    - launching, 92
    - layouts, inspecting, 220
    - modes, 92
      - layout controls above application content, 92
    - Layout View, 92–93
    - Pixel Perfect, 94
      - switching, 92
    - online tutorial, 94
    - overview, 91
    - user interface optimization, 94
  - hint attribute (EditText control), 177**
  - history of mobile devices**
    - Android manufacturers, 20
    - Android project, 20
    - device convergence, 17
    - first Android phone, 20
    - first cell phone, 13
    - first-generation, 13
    - form factors, 15
    - functionality, 11–13
    - Google Internet model, 19
    - Open Handset Alliance (OHA), 20
      - manufacturers, 20
      - mobile operators, 21
      - website, 35
    - operators, 21
  - platform market penetration, 18–19
  - proprietary platforms, 17
  - time-waster games, 14
  - WAP, 15–16
  - Home screen tips, 419**
  - Home section (documentation), 83**
  - HoneycombGallery sample app, 51**
  - horizontal progress indicators, 192**
  - HorizontalScrollView class, 229**
  - hprof-conv tool, 98**
  - HPROF files, creating, 429**
  - HTTP support package, 45**
  - hybrid development method, 329**
- 
- |
- <i> tag, 147**
  - icons**
    - alternative resources, 310
    - applications, configuring, 125, 380
    - emulator tips, 419
  - ICS (Ice Cream Sandwich), 5**
  - id fields, 223**
  - IDEs (integrated development environments), 25. See also Eclipse**
  - ImageButton class, 185**
  - images, 154**
    - accessing, 155
    - buttons, 185
    - drawables
      - accessing, 154
      - defining, 153
      - ShapeDrawable class, 153
    - formats, 154
    - GL texture compression formats, configuring, 131

- Nine-Patch Stretchable Graphics,
  - 95, 100
  - accessing, 155
  - creating, 95-97, 155
  - defined, 95, 155
  - device compatibility, 306
  - scaling, 95
  - storing, 139-141
- screen captures, 435
- ImageSwitcher controls, 230**
- importing**
  - classes (Eclipse), 73
  - code, 443
- improving workflow flexibility, 111-113**
- in-application billing, 379**
- inches, 152**
- incoming calls, simulating, 416, 434**
- indeterminate progress indicators, 193**
- indicators**
  - clocks
    - analog, 197
    - digital, 196
  - progress bars, 192-193
    - adjusting, 194
    - circular, 192
    - horizontal, 192
    - indeterminate, 193
    - title bars, adding, 193
  - ratings, 194-195
  - time passage, 195-196
- initializing**
  - activity data, 108-109
  - dialogs, 254
- input, 176**
  - autocomplete, 179-181
  - buttons, 183
    - basic, 183-184
    - check boxes, 183, 186
    - image buttons, 185
    - radio, 183, 187-189
    - switches, 183, 187
    - toggle, 183, 186
  - dates, 190-191
  - filters, 178-179
  - hints, 177
  - input box height, 177
  - retrieving, 176-178
  - specifying, 129
  - spinners, 181-183
  - text
    - alternative resource qualifiers, 315
    - contextual links, 174-176
    - displaying, 173-174
    - ellipsis, 174
    - height, 174
    - Toast messages, 185
    - width, 174
  - times, 191
- InputFilter interface, 178-179**
- inspecting**
  - layouts, 220
  - user interfaces at pixel level, 94
- installing**
  - development environment, 38
  - instructions website, 39
  - testing, 372
- Instrumentation tab (Eclipse), 122**

<integer> tag, 150  
**integers**, 150  
  accessing, 150  
  creating, 151  
  defining, 150  
  storing, 139  
**integrated development environments (IDEs)**, 25. *See also Eclipse*  
**integration**  
  applications, 26  
  points, testing, 371  
**intellectual property protection**, 378  
**Intent class**, 104  
**<intent-filter> tag**, 132  
**intents**, 104  
  action/data, 114  
  additional information, adding, 115  
  application navigation, 115–116  
  broadcasting, 117  
  filters, 114, 132–133  
  Google common, 115  
  launching activities  
    by class name, 113  
    external, 114  
  primary entry points, choosing, 132  
  website, 118  
**interfaces**  
  InputFilter, 178–179  
  OnChronometerTickListener, 196  
  SharedPreferences, 264  
    methods, 265  
    reference website, 273  
  SharedPreferences.Editor  
    methods, 266  
    reference website, 273  
  TabContentFactory, 226  
**internationalization, testing**, 372

INTERNET permission, 75  
**interoperability (applications)**, 342  
**isFinishing( ) method**, 111  
**ISO 3166-1-alpha-2 Regions website**, 322  
**ISO 639-1 Language codes website**, 322  
**Issue Tracker website**, 41  
**italic strings**, 147  
**iterative development**, 327, 346

---

**J**

**Java**  
  advantages, 26  
  autocomplete, 443  
  build errors, 447  
  classes, creating, 443  
  comments, 446  
  formatting, 444  
  imports, 443  
  methods, creating, 443  
  packages, 44, 284  
  perspective (Eclipse), 47  
  QuickFix feature, 446  
  refactoring code, 444–445  
  reorganizing, 446  
  website, 52  
**Javadoc comments**, 446  
**Java Development Kit (JDK)**, 37  
**JSON (JavaScript Object Notation) package**, 45  
**junit package**, 44

---

**K**

**key attribute**, 269  
**keyboard**  
  alternative resource qualifier, 314  
  support, 406

**killer application chances, increasing**, 374  
**killing activities**, 109-110

## L

---

**landscape-mode alternative picture graphic**, 316  
**languages**  
  alternative resource qualifier, 312  
  ISO 639-1 codes website, 322  
**last known location, finding**, 77  
**launch activities, creating**, 64  
**launch configurations**  
  creating  
    MyFirstAndroidApp, 66-67  
    Snake application, 58-59  
  Debug, 66  
  Run, 66  
**launching**  
  activities by class name, 113  
  applications, 114  
  AVD Manager, 47  
  dialogs, 255  
  emulator, 60  
    AVD Manager, 411  
    options, 407  
    specific projects, 408-409  
  Hierarchy Viewer, 92  
  SDK Manager, 47  
**layout\_gravity attribute**, 208  
**Layout View mode (Hierarchy Viewer)**, 92-93  
**layoutopt tool**, 220, 375  
**LayoutParams class**, 205  
**layouts**  
  attributes, 205-206  
  controls, 172  
  creating  
    programmatically, 201-203  
    XML, 199-200  
  declaring website, 231  
  Eclipse layout resource editor, 164-166  
  frames, 207-209  
  grids, 216-220  
  inspecting, 220  
  linear, 206, 209-210  
  multiple, 220  
  nesting, 207  
  relative, 211-214  
  resource files  
    accessing, 166  
    alternative versions, 167  
    background color/text example, 162-163  
    defined, 162  
    fragments, 244-246  
  storing, 140  
  tables, 214-216  
  types, 205  
**legacy support**, 6  
  contacts, 293-294  
  fragments, 247-248  
  package, 45  
  platform, 248  
**libraries**, 130  
**licensing**, 25  
  Agreement website, 52  
  End User License Agreement (EULA), 351  
  Google Market Licensing, 46  
  LVL, 378  
  SDK license agreement, 42-43

**lifecycles**

- activities, 106–107
- applications, 34
- dialogs, 254
- fragments, 235

**LinearLayout class, 172, 209–210, 231****lines attribute**

- EditText control, 177
- TextView control, 174

**links, creating, 174–176****Linux**

- configuration website, 39
- operating system, 30

**ListActivity class, 225, 231****listening preferences, 267****ListFragment class, 238**

- implementing, 240–243
- reference website, 250

**listing details (Android Market), 388****ListPreference class, 274****lists, 222**

- files, 278, 282
- fragments, 238
- user interfaces, creating, 225

**ListView class, 222, 231****live servers, managing, 345****local variables, extracting, 445****location-based services, adding**

- coordinates
  - sending, 418
  - website, 413
- emulator location, 77
- last known location, 77
- MyFirstAndroidApp, 76–78

**LocationManager class, 81****Log class**

- methods, 73
- website, 81

**logcat tool, 86–87, 99, 374, 436–437****logging**

- adding, 73–74
- filters, creating, 441
- turning off, 381

**loss of signal testing, 367****low-risk porting, 345****lowest common denominator development method, 327–328****LunarLander sample app, 51****LVL (License Verification Library), 378**

---

**M****maintenance**

- application diagnostics, leveraging, 353
- documentation, 338
- mobile development, 344–345

**managedQuery( ) method, 224, 286**

- Browser class, 290
- CallLog class, 289
- MediaStore.Audio.Media class, 288

**managers****AVD**

- launching, 47
- overview, 48

**SDK**

- launching, 47
- overview, 48

**managing**

- activity transitions
- action/data, 114
- additional information, adding, 115

- application navigation, 115-116
- broadcasting, 117
- filters, 114, 132-133
- Google common, 115
- launching activities, 113-114
- primary entry points, choosing, 132
- website, 118
- Android Market applications
  - removing, 393
  - return policies, 392
  - upgrades, 392
- AVDs, 402
- device databases, 330
  - data, storing, 332
  - devices for tracking, choosing, 332
  - functions, 332
  - third-party databases, 333
- files
  - best practices, 276-277
  - methods, 278-279
- live servers, 345
- memory allocations, 430
- testing environment, 365
  - clean states, 366-367
  - device configurations, 365-366
  - real world simulations, 367
- manifest files**
  - activities, registering, 131
  - intent filters, 132-133
  - primary entry points, 132
  - applications
    - enforced permissions, registering, 134
    - icons, 125
    - names, 125
    - versioning, 125
  - broadcast receivers, registering, 133
  - content providers, registering, 133
  - editing, 89, 120
    - application-wide settings, 121
    - Eclipse, 120
    - instrumentation settings, 122
    - manually, 123-124
    - package-wide settings, 121
    - permissions, 121
  - external libraries, 130
  - features, 135
  - GL texture compression formats, 131
  - names, 119
  - overview, 119-120
  - package names, 124
  - permissions
    - application enforced, registering, 134
    - groups, 135
    - protection levels, 135
    - required, registering, 133-134
  - platform requirements, enforcing, 129-130
    - device features, 129-130
    - input methods, 129
    - screen sizes, 130
  - publication preparations, 381
  - reference website, 136
  - screen compatibility, 131
  - SDK versions, targeting, 126-128
    - maximum, 128
    - minimum, 127
    - specifying, 128
  - services, registering, 133
  - system requirements, enforcing, 125-128

- Manifest tab (Eclipse), 121**
- manufacturers, 20, 334**
- map coordinates website, 413**
- MarginLayoutParams class, 205**
- Market (Android), 385, 392**
  - alternatives, 393-394
  - applications
    - removing, 393
    - upgrading, 392
    - uploading, 387
  - contact/consent options, 390
  - developer accounts, creating, 385-387
  - fees, 390
  - filters, 381, 395
  - help, 390
  - listing details, 388
  - marketing assets, uploading, 388
  - packaging requirements, 381
  - publishing options, 390
  - return policies, 392
  - reviews, 344
  - website, 7, 385, 395
- marketing. See distribution**
- markets**
  - focus, 23
  - targeting, 344
- mascot, 24**
- max VM application heap size, 407**
- maximizing, 367**
  - application
    - popularity, 374
    - upgrades, 371
    - usability, 370
  - automation, 368
  - backup services, 373
  - billing, 373
- black-box testing, 369**
- build validations, 368**
- compatibility, 303**
- conformance, 372**
- device upgrades, 372**
- emulators**
  - benefits, 368
  - disadvantages, 369
- installations, 372**
- integration points, 371**
- internationalization, 372**
- performance, 373**
- preproduction devices, 368**
- remote servers/services, 370**
- third-party standards, 371**
- unexpected configuration changes, 373**
- visual appeal, 370**
- white-box testing, 369**
- windows (Eclipse), 440**
- maxLines attribute, 174**
- measureAllChildren attribute, 208**
- measurements**
  - dimensions, 152
  - ems, 174
- media**
  - accessing, 286-288
  - adding, 74-76
  - audio file titles, retrieving, 287
  - queries, 288
- MediaPlayer class**
  - methods, 74
  - website, 81
- MediaStore content provider, 286-288**
  - audio file titles, retrieving, 287
  - classes, 286-287
  - website, 300

**memory allocations, 430**

**menus, 158**

- accessing, 159
- creating, 159
- defining, 158
- storing, 139

**messages**

- between emulators, 415
- Toast, 185

**methods**

- `addFooterView( )`, 226

- `addHeaderView( )`, 226

- `addPreferencesFromResource( )`, 271

- `addTab( )`, 227-229

- `addToBackStack( )`, 236

- `addView( )`, 204

- `addWord( )`, 291

- `apply( )`, 267

- `clear( )`, 266

- `clearCheck( )`, 189

- `commit( )`, 266

- `contains( )`, 265

- `create( )`, 75

- `createTabContent( )`, 228

- `creating`, 443

- `delete( )`, 299

- `deprecated`, 252

- `dismissDialog( )`, 254

- `edit( )`, 265

- `elapsedRealtime( )`, 196

- file management, 278-279

- `filter( )`, 179

- `forceError( )`, 69

- `Fragment` class, 237

- `getAll( )`, 265

- `getApplicationContext( )`, 104

- `getAssets( )`, 105

- `getBoolean( )`, 150, 265

- `getCacheDir( )`, 282

- `getColor( )`, 151

- `getContentResolver( )`, 298

- `getDimension( )`, 153

- `getExternalCacheDir( )`, 283-284

- `getExternalFilesDir( )`, 284

- `getExternalStoragePublicDirectory( )`,  
284

- `getExternalStorageState( )`, 284

- `getFloat( )`, 265

- `getInt( )`, 265

- `getLocation( )`, 78

- `getLong( )`, 265

- `getResources( )`, 105

- `getSharedPreferences( )`, 105

- `getString( )`, 266

- `getStringSet( )`, 266

- `getText( )`, 173, 178

- `isFinishing( )`, 111

- Log class, 73

- `managedQuery( )`, 224, 286

- Browser class, 290

- CallLog class, 289

- MediaStore.Audio.Media class, 288

- MediaPlayer class, 74

- `obtainTypedArray( )`, 162

- `onActivityCreated( )`, 237

- `onAttach( )`, 237

- `onCheckedChangedListener( )`, 189

- `onClick( )`, 185

- `onCreate( )`, 108, 237

- `onCreateDialog( )`, 254

- `onCreateView( )`, 237

- `onDestroy( )`, 110

onDestroyView( ), 237  
onDetach( ), 237  
onItemClick( ), 225  
onListItemClick( ), 225  
onPause( )  
    Activity class, 109  
    fragments, 237  
onPrepareDialog( ), 254  
onPressChanged( ), 194  
onRatingChanged( ), 195  
onResume( )  
    Activity class, 109  
    fragments, 237  
onRetainNonConfigurationInstance( ), 318  
onSaveInstanceState( ), 110  
onStart( ), 237  
onStop( ), 76, 237  
openFileOutput( ), 279  
parse( ), 75  
playMusicFromWeb( ), 75  
putBoolean( ), 266  
putFloat( ), 266  
putInt( ), 266  
putLong( ), 266  
putString( ), 266  
putStringSet( ), 266  
registerOnSharedPreferenceChangeListener( ), 267  
release( ), 75  
remove( ), 266  
removeDialog( ), 254  
set, 229  
setAdapter( ), 224  
setContentView( ), 173  
setFilters( ), 179  
setIndicator( ), 229  
setListAdapter( ), 225  
setOnItemClickListener( ), 185  
setOnItemClickListener( ), 224  
setOnTimeChangedListener( ), 191  
setText( ), 173, 178  
setup( ), 229  
SharedPreferences interface, 265  
SharedPreferences.Editor, 266  
showDialog( ), 254  
start( )  
    Chronometer control, 196  
    MediaPlayer class, 75  
stop( )  
    Chronometer control, 196  
    MediaPlayer class, 75  
unregisterOnSharedPreferenceChangeListener( ), 267  
Uri class, 74  
**millimeters, 152**  
**minimizing windows (Eclipse), 440**  
**minLines attribute, 174**  
**mistakes**  
    avoiding, 355, 360  
    testing, 375  
**mixed-type arrays, storing, 139**  
**mksdcard tool, 99**  
**mnt/sdcard/ directory, 432**  
**mnt/sdcard/download/ directory, 432**  
**mobile country code alternative resource qualifier, 312**  
**mobile development, 325**  
    applications  
        feasibility assessments, 336  
        interoperability, 342

best practices, 355  
code diagnostics, 358–359  
code reviews, 358  
coding standards, 357  
feasibility, testing, 356  
mistakes, avoiding, 360  
reference websites, 361  
single device bugs, 359  
software process, choosing, 356  
tools, leveraging, 360  
billing/revenue generation, 351–352  
    in-app, 379  
    testing, 373  
challenges, 325  
devices  
    availability, 334–335  
    databases, 330–333  
    limitations, 340  
diagnostics  
    anonymous, gathering, 354  
    code, 358–359  
    leveraging, 353  
distribution, 25, 343–344, 385. *See also publication*  
    ad revenue, 379  
    alternatives, 393–394  
    Android Market, 385–393  
    choosing, 377–378  
    in-application billing, 379  
    intellectual property protection, 378  
    market reviews, 344  
    options, 27–28  
    self-distribution, 394–395  
    statistics collection, 379  
documentation, 337–338  
extensibility, 341–342  
implementation steps, 342  
iteration, 327  
maintenance/support, 344–345  
manufacturers/operators risks, 334  
mistakes, avoiding, 355  
network-driven applications, 340  
outside influences, 335  
personal device testing, 330  
quality assurance, 336–338  
requirements, 327–330  
risk assessment, 333  
rules, 347–348  
security, 351  
source control systems, choosing, 339  
stability/responsiveness, 349–350  
standalone applications, 340  
target devices, 335  
testing, 343  
third-party standards, 352  
tools, 354  
updates/upgrades, 354  
user demands, meeting, 348  
user interfaces, 348–349  
versioning schemes, choosing, 339–340  
waterfall approaches, 326

**mobile network code alternative resource qualifier, 312**

**mobile operators, 21**

**Mobiletuts+ website, 7**

**monkey tool, 99**

**monkeyrunner tool, 99**

**MoreAsserts class, 359**

**Motorola DynaTAC 8000X, 13**

**MP3 playback, adding, 75**

- MultiAutoCompleteTextView control, 181**
- multiple layouts, 220**
  - multiple screens, 136, 322**
  - MyFirstAndroidApp**
    - AVD, creating, 65
    - core files, 65
    - debugging
      - emulator, 69–73
      - devices, 78–79
    - launch configuration, 66–67
    - location-based services, 76–78
    - logging support, 73–74
    - media support, 74–76
    - new project, creating, 62–64
      - application information, 64
      - build targets, 63
      - file locations, choosing, 63
      - launch activities, 64
      - minimum SDK version, 64
      - names, 62
      - package names, 64
      - running, 67–68

---

## N

- names**
  - applications, configuring, 125, 380
  - manifest file, 119
  - packages, 64, 124
  - projects, 62
  - SDKs, 25
- native versus third-party applications, 26, 33**
- navigation**
  - applications, 115–116
  - methods alternative resource qualifier, 315

- nesting, 207**
- network-driven applications, 340**
- network status, displaying, 418**
- New, Android Project command (File menu), 62**
- New, Other command (File menu), 54**
- night mode, 313**
- Nine-Patch Stretchable Graphics, 95**
  - accessing, 155
  - creating, 95–97, 155
  - defined, 95, 155
  - device compatibility, 306
  - scaling, 95
  - website, 100
- normal protection level (permissions), 135**
- NotePad sample app, 51**

---

## O

- obtainTypedArray( ) method, 162**
- Official Android Developers Blog, 35**
- OHA (Open Handset Alliance), 20**
  - manufacturers, 20
  - mobile operators, 21
  - website, 35
- onActivityCreated( ) method, 237**
- onAttach( ) method, 237**
- onCheckedChangedListener( ) method, 189**
- OnChronometerTickListener interface, 196**
- onClick( ) method, 185**
- onCreate( ) method, 108, 237**
- onCreateDialog( ) method, 254**
- onCreateView( ) method, 237**
- onDestroy( ) method, 110**
- onDestroyView( ) method, 237**
- onDetach( ) method, 237**

**onItemClick( ) method**, 225  
**OnItemClickListener class**, 224  
**onListItemClick( ) method**, 225  
**onPause( ) method**  
    Activity class, 109  
    fragments, 237  
**onPrepareDialog( ) method**, 254  
**onProgressChanged( ) method**, 194  
**onRatingChanged( ) method**, 195  
**onResume( ) method**  
    Activity class, 109  
    fragments, 237  
**onRetainNonConfigurationInstance( ) method**, 318  
**onSaveInstanceState( ) method**, 110  
**onStart( ) method**, 237  
**onStop( ) method**, 76, 237  
**Open Handset Alliance**. See OHA  
**open platform**, 24  
**open-source platforms**, 25  
**openFileInput( ) method**, 279  
**openFileOutput( ) method**, 279  
**OpenGL compatibility**, 301  
**opening files**, 279  
**operating systems**  
    application interaction, 34  
    configuring for device debugging, 39  
    supported, 25  
**operators**, 334  
**optimizing user interfaces**, 94  
**org.apache.http package**, 45  
**org.json package**, 45  
**org.w3c.dom package**, 45  
**org.xml.sax package**, 45  
**org.xmlpull package**, 45

**organizing activities (fragments)**  
    Activity classes, creating, 246–247  
    applications, designing, 238  
    attaching, 237  
    compatibility, 305  
    creating, 237  
    defined, 104, 111  
    defining, 235  
    destroying, 237  
    detaching, 237  
    dialogs, 238, 257–260  
    layout resource files, creating, 244–246  
    legacy support, 247–248  
    lifecycle, 235  
    ListFragment, implementing, 240–243  
    lists, 238  
    MP3 music player example, 111–113  
    overview, 233  
    pausing, 237  
    resuming activity, 237  
    screen workflow, 234  
    specialty, 237  
    stopping activity, 237  
    updates, 236  
    user preferences, 238  
    visibility, 237  
    web content, 238  
    website, 118  
    WebView, implementing, 243–244  
    workflow flexibility, improving,  
        111–113

**orientation, screens**  
    alternative resources, 316  
    directory qualifier, 313

**orientation attribute**  
    GridLayout class, 217  
    LinearLayout class, 210

---

**P****packages**

- android, 44
  - C2DM, 45
  - commonly used, 34
  - dalvik, 44
  - Google
    - Analytics SDK, 45
    - APIs, 45
    - Market Billing, 46
    - Market Licensing, 46
  - important, 43
  - java, 44
  - java.io, 284
  - javax, 44
  - junit, 44
  - legacy support, 45
  - manifest file settings, 121
  - names, 64, 124
  - org.apache.http, 45
  - org.json, 45
  - org.w3c.dom, 45
  - org.xml.sax, 45
  - org.xmlpull, 45
  - provider, 285, 300
  - Support (Compatibility)
    - adding, 248-249, 305
    - website, 118
  - test, 359
  - third-party APIs, 45
  - view, 171
  - widget, 171
- packaging**
- code preparations, 380
  - Android Market, 381
  - application names/icons, 380

- debugging/logging, turning off, 381
- manifest file, 381
- permissions, 382
- target platforms, 381
- versioning, 380
- publication requirements, 380
- release versions, testing, 384
- signing, 382-384

**pages, adding, 419****ParisView project, 165****parse( ) method (Uri class), 75****PeakBagger 1.0, 86****performance**

- alternative resources, 317
- emulator, 407-408
- testing, 373

**PerformanceTestCase class, 359****<permission> tag, 135****permissions**

- ad-hoc, 31
- adding, 75
- application specific, 31
- contacts, 292
- content providers, 289
- files, 277
- groups, 135
- INTERNET, 75
- manifest file settings, 121
- protection levels, 135
- registering, 133-134
- settings, 292
- verifying, 382
- voicemail, 291
- website, 136

**Permissions tab (Eclipse), 121****personal devices, testing, 330**

**perspectives (Eclipse), 440, 47**

**Pixel Perfect mode (Hierarchy Viewer), 92-94**

**pixels, 152, 314**

**platform**

- advantage over competitors, 25
- architecture, 29
- Dalvik VM, 31
- Linux operating system, 30
- compatibility, 301
- complete, 23
- definition, 23
- differences
  - application development, 26
  - application integration, 26
  - development tools, 25
  - distribution options, 27
  - growing platform, 28
  - IDEs, 25
  - Java, 26
  - open-source, 25
  - publication, 27
  - SDKs, 25
  - free, 24
  - legacy support, 248
  - market penetration, 18-19
  - open, 24
  - open-source, 25
  - proprietary, 17
  - requirements
    - device features, 129-130
    - enforcing, 129-130
    - input methods, 129
    - screen sizes, 130
  - security, 31
    - ad-hoc permissions, 31
    - application-specific permissions, 31
- applications running as operating system users, 31
- certificate signing, 32
- developer registration, 32
- target, 328, 381

**playMusicFromWeb( ) method, 75**

**plug-ins, ADT Eclipse, 46-47, 88**

**resources**

- creating, 143-146
- editors, 89
- layout resource editor, 164-166
- UI designer, 90

**plural strings, 149**

**points, 152**

**popular applications, creating, 374**

**porting**

- documentation, 338
- low-risk, 345

**portrait-mode alternative picture graphic, 316**

**power settings, simulating, 418**

**pre-publication checklist website, 361**

**Preference class, 269**

- attributes, 269
- reference website, 274

**PreferenceActivity class, 268-273**

**PreferenceCategory class, 274**

**PreferenceFragment class, 238, 250**

**preferences**

- accessing, 105, 267-268
- adding, 264
- appropriateness, 263
- data types supported, 264
- editing, 266-267
- overview, 263
- PreferenceActivity class, 268
- private, 264

- reacting, 267
- reading, 265
- searching, 265
- shared, 265
- user
  - fragments, 238
  - loading up, 270-272
  - resource file, creating, 269-270
- PreferenceScreen class, 274**
- <PreferenceScreen> tag, 269**
- preproduction devices, 368**
- primary entry points (activities), 132**
- primitive resources, storing, 140**
- private data, 277, 351**
- private preferences, 264**
- processes, stopping, 426**
- profiling user interfaces, 100**
- programming language choices, 32-33**
- progress**
  - bars, 192-193
    - circular, 192
    - horizontal, 192
    - indeterminate, 193
    - title bars, adding, 193
  - dialogs, 253
- ProgressBar class, 192-193**
- ProgressDialog class, 253, 260**
- proguard.cfg file, 65**
- project.properties file, 65**
- projects**
  - application feasibility assessments, 336
  - creating, 62-64
    - application information, 64
    - build targets, 63
    - file locations, choosing, 63
    - launch activities, 64
- minimum SDK version, 64
- names, 62
- package names, 64
- device databases, 330
  - adapters, 223-224
  - data, storing, 332
  - devices for tracking, choosing, 332
  - functions, 332
  - third-party, 333
- documentation, 337-338
- manufacturers/operators, 334
- outside influences, 334-335
- ParisView, 165
- quality assurance, 336-337
- requirements, 327
  - customization, 328-329
  - hybrid approaches, 329
  - lowest common denominator, 327-328
  - third-parties, 330
  - use cases, creating, 329
- risk assessment, 333
- sample, 52
- source control systems, 339
- target devices
  - acquiring, 335
  - identifying, 333
  - versioning schemes, 339-340
- prompt attribute, 183**
- proprietary platforms emergence, 17**
- protection levels (permissions), 135**
- protocols**
  - HTTP support package, 45
  - WAP, 15-16
- provider package, 285, 300**
- <provider> tag, 133**

**publication**

- ad revenue, 379
- advantages, 27
- alternatives, 393–394
- Android Market, 385, 392
  - contact/consent options, 390
  - developer accounts, creating, 385–387
  - fees, 390
  - help, 390
  - listing details, 388
  - marketing assets, uploading, 388
  - publishing options, 390
  - removing applications, 393
  - return policies, 392
  - upgrading applications, 392
  - uploading applications, 387
  - website, 385
- code preparation, 380
  - Android Market, 381
  - application names/icons, 380
  - debugging/logging, turning off, 381
  - manifest file, 381
  - permissions, 382
  - target platforms, 381
  - versioning, 380
- distribution methods, 377–378, 385
- in-application billing, 379
- intellectual property protection, 378
- pre-publication checklist website, 361
- release versions, testing, 384
- requirements, 380
- self-publishing, 394–395
- signing applications, 382–384
- statistics collection, 379

**putBoolean( ) method, 266****putFloat( ) method, 266****putInt( ) method, 266****putLong( ) method, 266****putString( ) method, 266****putStringSet( ) method, 266**

---

**Q****qualifiers (directory), alternative resources, 309**

- bad examples, 311
- case, 310
- combining, 310
- customizing, 311
- default resources, including, 311
- good examples, 311
- limits, 310
- listing of, 311–313
- requirements, 311

**quality assurance**

- third-party standards, 352
- risks, 336
  - clients/servers/cloud, 337
  - device tests, 336
  - real-world testing limitations, 337
  - test hardware, obtaining, 336
  - test plans, 338

**querying**

- browsers, 290
- call logs, 289
- contacts
  - Contacts provider, 293–294
  - ContactsContract content provider, 295
  - quickly, 294–295
- media, 288

**QuickFix feature, 446**

## R

---

- RAD (Rapid Application Development), 346**
- radio buttons, 187-189**
- RadioButton class, 183, 187-189**
- RadioGroup class, 187-189**
- RAM size (devices), 405**
- RatingBar class, 194-195**
- ratings, displaying, 194-195**
- raw files, 160**
  - accessing, 161
  - defining, 160
  - reading, 280
  - storing, 140
- reading**
  - files, 277
    - byte-by-byte, 280
    - default application directories, 280
    - XML files, 280-281
  - preferences, 265
- real world testing simulations, 367**
- records (content providers)**
  - adding, 297-298
  - deleting, 298-299
  - updating, 298
- refactoring code, 444-445**
- Reference section (documentation), 84**
- referencing resources, 161**
  - aliases, 162
  - format, 161
  - string example, 161
- region code alternative resource qualifier, 312**
- registering**
  - activities, 131
    - intent filters, 132-133
    - primary entry points, 132
- broadcast receivers, 133
- content providers, 133
- permissions, 133-134
- services, 133
- registerOnSharedPreferenceChange Listener( ) method, 267**
- RelativeLayout class, 211-214, 231**
- release versions, testing, 384**
- release( ) method, 75**
- releasing activity data, 109**
- remove( ) method, 266**
- removeDialog( ) method, 254**
- Rename tool, 444**
- reorganizing code, 446**
- requirements**
  - Android Market packaging, 381
  - permissions, 133-134
  - platform
    - device features, 129-130
    - enforcing, 129-130
    - input methods, 129
    - screen sizes, 130
- projects, 327**
  - customization, 328-329
  - hybrid approaches, 329
  - lowest common denominator, 327-328
  - third-parties, 330
  - use cases, creating, 329
- publication, 380**
- software, 37-38**
- system, 125-128**
- res folder, 65**
- res/drawable file, 65**
- res/layout/main.xml file, 65**
- res/values/strings.xml file, 65**

**Resource editor, 89**

- layouts, designing, 164–166
- manifest file, 89

**resources**

- accessing, 142
- aliases, 162
- alternative, 141–142
  - benefits, 308
  - creating, 309
  - data, saving, 318
  - defined, 308
  - directory qualifiers, 309–313
  - icons example, 310
  - organization schemes, 317
  - performance issues, 317
  - requesting, 316
  - runtime changes, handling, 318
  - screen orientations, 316
  - website, 321
- animations
  - storing, 139
  - tweened, 156
  - types, 156
- application, retrieving, 105
- Boolean, 149
  - accessing, 150
  - defining, 150
  - storing, 139
- colors, 151
  - accessing, 151
  - defining, 151
  - formats, 151
  - storing, 139
- creating, 143–146
- default, 141–142, 308
- defined, 137
- dimensions, 152
  - accessing, 153
  - defining, 152
  - resource file example, 152
  - storing, 139
  - unit measurements, 152
- directories, 138
- drawables
  - accessing, 154
  - defining, 153
  - ShapeDrawable class, 153
  - storing, 139
- frame-by-frame animations, 156–157
- images, 154
  - accessing, 155
  - formats, 154
  - nine-patch stretchable, 155
  - storing, 139–141
- integers, 150
  - accessing, 150
  - arrays, 151
  - defining, 150
  - storing, 139
- layouts
  - accessing, 166
  - alternative versions, 167
  - background color/text example, 162–163
  - creating, 199–200
  - defined, 162
  - Eclipse layout resource editor, 164–166
  - fragments, creating, 244–246
  - storing, 140

- menus, 158
  - accessing, 159
  - creating, 159
  - defining, 158
  - storing, 139
- mixed-type arrays, 139
- primitive, 140
- raw files, 160
  - accessing, 161
  - defining, 160
  - storing, 140
- references, 161-162
- selectors, 156
- storing, 137-140
- strings
  - accessing, 147-148
  - arrays, 149
  - bold/italic/underline, 147
  - formatting, 146-147
  - overview, 146
  - plural, 149
  - storing, 139
  - <string> tag, 146
- styles, storing, 140
- system, 167-168
- themes, storing, 140
- tweened animations
  - accessing, 158
  - defining, 157
  - overview, 157
  - spinning graphic example, 158
- types, 138-140
- user preferences
  - creating, 269-270
  - loading up, 270-272
- websites, 168
- XML files, 159
  - accessing, 160
  - defining, 160
  - storing, 139
- Resources section (documentation), 84**
- responsiveness, 349-350, 361**
- retrieving**
  - activity data, 109
  - application resources, 105
  - assets, 105
  - audio file titles, 287
  - cache subdirectory, 279
  - context, 104
  - file handles, 278
  - files subdirectory, 279
  - subdirectories, 279
- return policies (Android Market), 392**
- revenue/billing generation, 351-352**
  - in-app, 379
  - testing, 373
- risk assessment, 333**
  - application feasibility, 336
  - device availability, 334-335
  - manufacturers/operators, 334
  - quality assurance, 336-337
  - clients/servers/cloud, 337
  - device tests, 336
  - real-world testing limitations, 337
  - test hardware, obtaining, 336
- target devices**
  - acquiring, 335
  - identifying, 333
- row attribute, 218**
- rowCount attribute, 217**
- rowSpan attribute, 218**
- rs:ResEnum application, 168**

**Rubin, Andy, 20**

**rules (design), 347-348**

**run configurations**

creating, 409

MyFirstAndroidApp, 66

**Run Configurations command**

(Run menu), 66

**Run menu commands**

Debug Configurations, 58, 66

Run Configurations, 66

**running applications**

MyFirstAndroidApp, 66-68

runtime changes, handling, 321

Snake, 59-60

## S

---

**sample applications, 50-52**

**saving**

activities

data, 109

state, 110

alternative resource data, 318

**scale-independent pixels, 152**

**screens**

aspect ratio alternative resource qualifier, 313

captures, 435

compatibility

devices, 305-307

fragments, 305

mode, 306, 321

nine-patch stretchable graphics, 306

screen type support, 305-306

settings, 131

Support (Compatibility)

Package, 305

user interfaces, 303-304

density-independent pixels, 152

dimensions alternative resource qualifier, 312

information, accessing, 304

multiple, 136, 322

orientation, 313, 316

pixel density alternative resource qualifier, 314

scrolling, 229

size

alternative resource qualifier, 313

compatibility, 301

configuring, 130

touch types, 314

workflow, 234

website, 130

**scrolling screens, 229**

**ScrollView control, 229**

**SDK (Software Development Kit)**

advantages, 25

application framework

important packages, 43-45

third-party APIs, 45

documentation, 43, 83-85

download website, 52

emulator, launching, 411

license agreement, 42-43, 52

Manager

launching, 47

overview, 48

names, 25

problems, 41

versions, targeting, 126-128

maximum, 128

minimum, 127

specifying, 128

upgrades, 29, 41

**seamlessness design website, 361**

**searching**

Eclipse, 442

preferences, 265

**SearchRecentSuggestions content provider, 286, 300**

**security, 31**

applications running as operating system users, 31

certificate signing, 32

designing, 351

developer registration, 32

permissions

ad-hoc, 31

adding, 75

application specific, 31

contacts, 292

content providers, 289

files, 277

groups, 135

INTERNET, 75

manifest file settings, 121

protection levels, 135

registering, 133-134

settings, 292

verifying, 382

voicemail, 291

website, 136

website, 136

**seek bars, 194**

**SeekBar class, 194**

**<selector> tag, 156**

**selectors, 156**

**self-publishing applications, 394-395**

**servers**

live, managing, 345

testing, 337, 370

**Service class, 104, 116**

**services**

backup, testing, 373

defined, 104

location-based, adding, 76-78

coordinates website, 413

emulator location, 77

last known location, 77

MyFirstAndroidApp, 76-78

sending coordinates, 418

overview, 116

purposes, 116

registering, 133

source control, 439

testing, 370

**ServiceTestCase class, 359**

**setAdapter( ) method, 224**

**setContentView( ) method, 173**

**setCurrentTabByTag( ) method, 229**

**setFilters( ) method, 179**

**setIndicator( ) method, 229**

**setListAdapter( ) method, 225**

**setOnClickListener( ) method, 185**

**setOnItemClickListener( ) method, 224**

**setOnTimeChangedListener( ) method, 191**

**setText( ) method, 173, 178**

**Settings content provider, 286, 292, 300**

**setup( ) method, 229**

**ShapeDrawable class, 153**

**shared preferences, creating, 265**

**SharedPreferences class, 105, 264**

**SharedPreferences interface**, 264

  methods, 265

  reference website, 273

**SharedPreferences.Editor interface**

  methods, 266

  reference website, 273

**SHOP4APPS website**, 393

**showDialog( ) method**, 254

**shrinkColumns attribute**, 215

**signature protection level (permissions)**, 135

**signing applications**, 382-384

**SimpleFragDialogActivity class**, 258

**size**

  cache partitions, 407

  max VM application heap, 407

  progress indicators, 192

  RAM (devices), 405

  screens

    alternative resource qualifier, 313

    compatibility, 301

    configuring, 130

**sliding drawers**, 230

**SlidingDrawer class**, 230

**smoke testing**, 368

**SMS messages, simulating**, 416, 434

**Snake app**, 51

  adding to Eclipse, 54

  AVD, creating, 56-57

  launch configuration, creating, 58-59

  missing folder error, 56

  running in emulator, 59-60

  sample code website, 81

**Soc.io Mall marketplace**, 393

**software**

  development kit. *See* SDK

  processes, choosing, 356

  iterative, 327

  waterfall approaches, 326

  website, 346

  requirements, 37-38

  upgrades, 23

**source code websites**, 6

**source control**, 339, 439

**span attribute**, 215

**specialized testing**

  application popularity, increasing, 374

  backup services, 373

  billing, 373

  conformance, 372

  installations, 372

  integration points, 371

  internationalization, 372

  performance, 373

  unexpected configuration changes, 373

  upgrades

    applications, 371

    devices, 372

**specialty fragments**, 237

**Spinner class**, 181-183

**Spinner sample app**, 51

**SpinnerTest sample app**, 51

**sqlite3 tool**, 99, 375

**src folder**, 65

**src/com/androidbook/myfirstandroidapp/  
  MyFirstAndroidAppActivity.java activity**, 65

**stability**, 349-350

**Stack Overflow website**, 6

**stacks (activities), 107**

**standalone applications, 340**

**start( ) method**

- Chronometer control, 196
- MediaPlayer class, 75

**startup options (emulator), 408**

**state, activity callbacks, 107-109**

- initializing/retrieving activities, 109
- static data, initializing, 108
- stopping/saving/releasing, 109

**static activity data, initializing, 108**

**statistics collection, adding, 379**

**stepping through code (Eclipse), 72**

**stop( ) method**

- Chronometer control, 196
- MediaPlayer class, 75

**stopping**

- activity data, 109
- processes, 426

**storing**

- animations, 139
- application data, 275-276
- Booleans, 139
- cache files, 282-283
- capacity, 276
- colors, 139
- device data, 332
- dimensions, 139
- drawables, 139
- files, 141, 283-284
- images, 139-141
- integers, 139
- layouts, 140
- menus, 139
- mixed-type arrays, 139
- primitive resources, 140

**private data, 351**

**raw files, 140**

**resources, 137-140**

- strings, 139
- styles, 140
- themes, 140
- XML files, 139

**stretchColumns attribute, 215**

**StrictMode class, 373**

**<string> tag, 146**

**strings, 149**

- accessing, 147-148
- bold/italic/underline, 147
- formatting, 146-147
- overview, 146
- plural, 149
- storing, 139

**styles, storing, 140**

**subdirectories**

- cache, 279
- creating, 282
- files, retrieving, 279
- retrieving, 279

**summary attribute, 269**

**Support4Demos app, 51**

**Support13Demos app, 51**

**<supports-gl-texture> tag, 131**

**<supports-screen> tag, 306**

**Switch control, 183, 187**

**switches, 183, 187, 230**

**system**

- requirements
- enforcing, 125-128
- SDK versions, targeting, 126-128
- resources, 167-168

# T

---

**T-Mobile G1, 20**

**tab interfaces, 226**

TabActivity class, 226-228

TabHost class, 229

**TabActivity class, 226-228, 231, 238**

**TabContentFactory interface, 226**

**TabHost class, 229-231**

**TableLayout class, 172, 214-216, 231**

**tables, 214-220**

**tablet compatibility, 318-319**

**tabs (Eclipse), closing, 441**

**TabSpecs class, 226**

**tags**

<animation-list>, 157

<b>, 147

<color>, 151

<compatible-screens>, 131

<dimen>, 152

<drawable>, 153-154

<fragment>, 235

<i>, 147

<integer>, 150

<intent-filter>, 132

<permission>, 135

<PreferenceScreen>, 269

<provider>, 133

<selector>, 156

<string>, 146

<supports-gl-texture>, 131

<supports-screens>, 306

<TextView>, 173

<u>, 147

<uses-configuration>, 129

<uses-feature>, 129-130

<uses-library>, 130

<uses-permission>, 134

**target devices**

acquiring, 335

identifying, 333

**targeting**

markets, 344

platforms, 328, 381

SDK versions, 126-128

maximum, 128

minimum, 127

specifying, 128

**tasks (Eclipse), 442**

**test package, 359**

**testing**

applications, 343

popularity, increasing, 374

usability, 370

automating, 368

backup services, 373

billing, 373

black-box, 369

build validations, 368

clients/servers/cloud, 337

conformance, 372

content providers, 285

coverage, maximizing, 367

defect tracking systems, 363-365

development environment with Snake application, 53

adding to Eclipse, 54

AVD, creating, 56-57

launch configuration, creating, 58-59

running in emulator, 59-60

devices, 336

emulator  
 benefits, 368  
 disadvantages, 369  
 environment, 365  
 clean states, 366–367  
 device configurations, 365–366  
 real world simulations, 367  
 feasibility, 356  
 firmware upgrades, 345  
 hardware, obtaining, 336  
 installations, 372  
 integration points, 371  
 internationalization, 372  
 loss of signal, 367  
 mistakes, 375  
 performance, 373  
 personal devices, 330  
 plans, 338  
 preproduction devices, 368  
 real-world limitations, 337  
 reference websites, 376  
 release versions, 384  
 remote servers/services, 370  
 third-party  
   documentation, 338  
   firmware modifications, 365  
   standards, 371  
 tools, 374–375  
 unexpected configuration changes, 373  
 unit, 358  
 upgrades  
   applications, 371  
   devices, 372  
 visual appeal, 370  
 white-box, 369

**text**  
 contextual links, 174–176  
 displaying, 173–174  
 ellipsis (...), 174  
 height, 174  
 input methods alternative resource qualifier, 315  
 Toast messages, 185  
 width, 174

**TextSwitcher class, 230**

**TextView class, 173**  
 attributes, 173–174  
 contextual links, 174–176  
 height, 174  
 width, 174

**<TextView> tag, 173**

**themes, 140**

**third-party**  
 APIs, 45  
 content providers, 299  
 device databases, 333  
 firmware modifications, 365  
 native applications, compared, 33  
 quality standards, 352  
 requirements, 330  
 standards, 371  
 testing documentation, 338

**threads, monitoring, 426**

**TicTacToeLib sample app, 51**

**TicTacToeMain sample app, 51**

**time-waster games appearances, 14**

**TimePicker class, 191**

**TimePickerDialog class, 253, 260**

**times**

- clocks**
  - analog, 197
  - digital, 196
- picker dialogs**, 253
- timers**, 195–196
- user input**, 191

**title attribute**, 269

**title bars (progress indicators)**, 193

**Toast messages**, 185

**ToggleButton control**, 183, 186

**toggles**, 183, 186

**toLeftOf attribute**, 213

**tools**, 25, 46

- ADB, 87
- Android documentation, 83–85
- application design, 354
- AVD Manager, 47–48
- bmgr, 98, 375
- DDMS
  - debuggers, attaching, 425
  - debugging applications, 87
  - drag-and-drop operations, 433
  - Emulator Control pane, 434
  - features, 424
  - File Explorer, 430–433
  - garbage collection, 428
  - heap statistics, 427
  - HPROF files, 429
  - logging, 436–437
  - memory allocations, 430
  - perspective (Eclipse), 47
  - processes, stopping, 426
  - screen captures, 435
  - standalone application, 423
- threads, monitoring, 426
- website, 100

- development, leveraging, 360
- dmtracedump, 98
- Eclipse ADT plug-in, 46–47, 88
- resources, 89, 143–146, 164–166
- UI designer, 90
- emulator, 49
- AVDs, 401–407
- benefits, 368
- best practices, 399–401
- calling between, 413
- console, 416–419
- Control pane, 434
- disadvantages, 369
- files, managing, 432–433
- GPS location, 411–413
- Hierarchy Viewer, launching, 92
- icon tips, 419
- launching, 60, 407–411
- limitations, 420–421
- location, configuring, 77
- messaging between, 415
- MyFirstAndroidApp, 67–73
- overview, 85, 399
- pages, adding, 419
- PeakBagger 1.0, 86
- performance, 407–408
- screen captures, 435
- smartphone-style example, 49
- Snake app, running, 59–60
- startup options, 408
- tablet-style example, 50
- tips, 419
- unlocking, 60

- wallpaper, editing, 419
  - website, 86, 100, 421
  - etc1tool, 99
  - Exerciser Monkey, 373, 376
  - Extract Local Variable, 445
  - Extract Method, 445
  - Hierarchy Viewer
    - launching, 92
    - layout controls above application content, 92
    - Layout View mode, 92–93
    - modes, 92
    - online tutorial, 94
    - overview, 91
    - Pixel Perfect mode, 94
    - user interface optimization, 94
  - hprof-conv, 98
  - layoutopt, 220, 375
  - logcat, 99, 374, 436–437
  - mksdcard, 99
  - monkey, 99
  - monkeyrunner, 99
  - Nine-Patch Stretchable Graphics, 95–97
  - Rename, 444
  - resource editors, 89
  - sample applications, 50–51
  - SDK Manager, 47–48
  - signing, 384
  - sqlite3, 99, 375
  - testing, 374–375
  - traceview, 98, 375
  - UI designer, 90
  - website, 98
  - zipalign, 99
- toRightOf attribute, 213**
  - touch screen support, 405, 314**
  - TouchUtils class, 359**
  - traceview tool, 98, 375**
  - trackball support, 406**
  - tracking**
    - devices, 332
    - memory allocations, 430
  - transitions (intents), 104**
    - action/data, 114
    - additional information, adding, 115
    - application navigation, 115–116
    - broadcasting, 117
    - filters, 114, 132–133
    - Google common, 115
    - launching activities
      - by class name, 113
      - external, 114
    - primary entry points, choosing, 132
    - website, 118
  - troubleshooting. See also debugging**
    - alternative resource performance issues, 317
    - build errors, 447
    - crashes, 345
    - layouts, 220
    - SDK problems, 41
    - single device bugs, 359
    - Snake application missing folder, 56
    - user interfaces
      - drawing issues, 92–93
      - layout control organization, 94
  - trust relationships, 32**
  - tweened animations, 156**
    - accessing, 158
    - defining, 157

overview, 157  
spinning graphic example, 158

**types**

- animations, 156
- buttons, 183
- defects, 364–365
- dialogs, 252–253
- fragments, 237
- layouts, 205
- resources, 138–140

## U

---

**<u> tag, 147**

**UI designer, 88–90**

**underlining strings, 147**

**unexpected configuration changes, testing, 373**

**Uniform Resource Identifiers (URIs), 31**

**unit testing, 358**

**unlocking, emulator, 60**

**unregisterOnSharedPreferenceChangeListener( ) method, 267**

**updating**

- contacts, 298
- designing for, 354
- fragments, 236
- preferences, 266–267

**upgrades, 23**

- applications, 371, 392
- designing for, 354
- devices, 372
- firmware, testing, 345
- SDKs, 41, 29

**uploading (Android Market)**

- applications, 387
- marketing assets, 388

**Uri class**

- methods, 74
- website, 81

**URIs (Uniform Resource Identifiers), 31**

**usability testing, 370**

**use cases, creating, 329**

**user-facing features, 23**

**user interfaces**

- autocomplete, 179–181
- buttons, 183
  - basic, 183–184
  - check boxes, 183, 186
  - image buttons, 185
  - radio buttons, 183, 187–189
  - switches, 183, 187
  - toggles, 183, 186
- compatibility, 303–304
  - best practices, 322
  - fragments, 305
  - nine-patch stretchable graphics, 306
  - screen types, 305–306
  - Support (Compatibility) Package, 305
  - working squares, 306–307
- containers, 221
- controls
  - App Widgets, compared, 172
  - defined, 171
  - layout, 172
- data-driven controls, 221–222
  - adapters, 222
  - arrays, 222–223
  - click events, handling, 224
  - data, binding, 224
  - databases, 223–224
  - headers/footers, 226
  - lists of items, 225

- date input, 190-191
- debugging and profiling website, 100
- designing, 348-349
- dialogs
  - adding, 251-254
  - alerts, 252
  - basic, 252
  - character pickers, 252
  - customizing, 256
  - date pickers, 253
  - defining, 254
  - destroying, 255-256
  - dismissing, 255
  - fragment method, 257-260
  - initializing, 254
  - launching, 255
  - lifecycle, 254
  - progress, 253
  - time pickers, 253
  - types, 252-253
  - websites, 260
- documentation, 338
- drawing issues, debugging, 92-93
- fragments
  - Activity classes, creating, 246-247
  - applications, designing, 238
  - attaching, 237
  - compatibility, 305
  - creating, 237
  - defined, 104, 111
  - defining, 235
  - destroying, 237
  - detaching, 237
  - layout resource files, creating, 244-246
- legacy support, 247-248
- lifecycle, 235
- ListFragment, implementing, 240-243
- lists, 238
- MP3 music player example, 111-113
- overview, 233
- pausing, 237
- resuming activity, 237
- screen workflow, 234
- specialty, 237
- stopping activity, 237
- updates, 236
- user preferences, 238
- visibility, 237
- web content, 238
- website, 118
- WebView, implementing, 243-244
- workflow flexibility, improving, 111-113
- galleries, 205, 222
- grids, 222
- guidelines website, 361
- indicators
  - adjusting progress, 194
  - clocks, 196-197
  - progress bars, 192-193
  - ratings, 194-195
  - time passage, 195-196
- inspecting at pixel level, 94
- layouts
  - attributes, 205-206
  - creating programmatically, 201-203
  - creating XML, 199-200

declaring website, 231  
frames, 207-209  
grids, 216-220  
inspecting, 220  
linear, 206, 209-210  
multiple, 220  
nesting, 207  
organization, 94  
relative, 211-214  
tables, 214-216  
types, 205  
lists, 222  
Nine-Patch Stretchable Graphics, 95  
accessing, 155  
creating, 95-97, 155  
defined, 95, 155  
device compatibility, 306  
scaling, 95  
website, 100  
optimizing, 94  
scrolling, 229  
sliding drawers, 230  
spinners, 181-183  
switchers, 230  
tabs, 226  
    TabActivity class, 226-228  
    TabHost control, 229  
text, displaying, 173  
    contextual links, 174-176  
    ellipsis, 174  
    height, 174  
    width, 174  
time input, 191  
user input  
    autocomplete, 179-181  
    dates, 190-191  
    filters, 178-179  
    hints, 177  
    input box height, 177  
    retrieving, 176-178  
    spinners, 181-183  
    times, 191  
View class, 171, 204  
ViewGroups, 204  
**UserDictionary content provider, 286, 291, 300**  
**users**  
    demands, meeting, 348  
    input, 176  
        autocomplete, 179-181  
        dates, 190-191  
        filters, 178-179  
        hints, 177  
        input box height, 177  
        retrieving, 176-178  
        spinners, 181-183  
        times, 191  
    preferences  
        fragments, 238  
        loading up, 270-272  
        resource files, creating, 269-270  
**<uses-configuration> tag, 129**  
**<uses-feature> tag, 129-130**  
**<uses-library> tag, 130**  
**<uses-permission> tag, 134**

**V****V CAST Apps website, 393****versioning**

- applications, 125
- code, 380
- schemes, choosing, 339–340

**Videos section (documentation), 84****View class, 171, 204**

- adding to ViewGroup, 204
- containers, 204
- galleries, 205
- grouping, 204

**View containers**

- sliding drawers, 230
- switchers, 230
- tabs, 228

**view package, 171****ViewAsserts class, 359****ViewGroup class**

- layout attributes, 205–206
- layout subclasses, 204
- view containers, 204
- View class, compared, 204
- website, 231

**views**

- AdapterView controls, 222
- containers, 221
- data-driven containers, 221–222
  - adapters, 222
  - arrays, 222–223
  - click events, handling, 224
  - data, binding, 224
  - databases, 223–224
  - lists of items, 225
- FrameLayout, 207–209

**GridLayout, 216–220****LinearLayout, 209–210****RelativeLayout, 211–214****scrolling support, 229****TableLayout, 214–216****tabs, 226–227****ViewSwitcher class, 230****virtual device management website, 421****visual appeal, testing, 370****VMs (virtual machines), 31****voicemail, accessing, 291****VoicemailContract content provider, 286, 291**

---

**W****wallpaper, editing, 419****WAP (Wireless Application Protocol), 15**

- browsers, 16

- commercializing, 16

- introduction, 15

**waterfall development, 326, 346****web**

- application development, 33
- content, 238

**websites**

- Activity class, 81, 118

- ad revenue, 379

- ADB, 88, 100

- Adobe AIR, 33

- AlarmClock content provider, 300

- AlertDialog class, 260

- alternative marketplaces, 394

- alternative resources, 321

- Amazon Appstore, 393

- anddev.org forum, 7

Android  
Developers, 6, 350  
Development, 35  
documentation, 83  
Tools Project Site, 7  
API levels, 136  
book, 6  
Browser content provider, 300  
CalendarContract content provider, 291  
CallLog content provider, 300  
CharacterPickerDialog class, 260  
CheckBoxPreference class, 274  
“Common Tasks and How to Do Them in Android,” 81  
compatibility  
best practices, 322  
package, 118  
<compatible-screens> tag, 131  
Conder blog, 8  
Contacts content provider, 300  
ContactsContract content provider, 297, 300  
Context class, 118, 284  
country codes, 322  
Darcey blog, 8  
DatePickerDialog class, 260  
DDMS, 100  
design best practices, 361  
Developer.com, 7  
“Developing on a Device,” 81  
development  
system requirements, 38  
tools, 98  
Dialog class, 260, 321  
DialogFragment class, 250, 260  
dialogs, 260  
Eclipse, 52  
EditTextPreference class, 274  
emulator, 86, 100, 421  
Environment class, 284  
Exerciser Monkey, 376  
extreme programming, 346  
FierceDeveloper, 7  
File class, 284  
Fragment class, 118, 250  
fragments, 118  
FrameLayout class, 231  
Gallery class, 231  
Google  
Analytics on Android, 361  
Android Developer’s Guide, 52, 100  
common intents, 115  
Experience devices, 335  
Maps, 413  
Team Android Apps, 7  
TV media formats, 321  
GridLayout class, 231  
GridView class, 231  
Handango, 393  
Hierarchy Viewer online tutorial, 94  
IDEs besides Eclipse, 38  
in-application billing, 379  
installation instructions, 39  
intents, 118  
Issue Tracker, 41  
iterative development, 346  
Java Platform, Standard Edition, 52

java.io package, 284  
language codes, 322  
layouts, declaring, 231  
LinearLayout class, 231  
Linux configuration, 39  
ListActivity class, 231  
ListFragment class, 250  
ListPreference class, 274  
ListView class, 231  
LocationManager, 81  
Log class, 81  
LVL, 378  
manifest file reference, 136  
Market, 7, 385, 395  
    content guidelines, 390  
    help, 390  
    fees, 390  
    filters, 381, 395  
    reviews, 344  
MediaPlayer class, 81  
MediaStore content provider, 300  
memory allocations, tracking, 430  
Mobiletuts+, 7  
multiple screens, 136, 322  
Nine-Patch Stretchable Graphics, 100  
Official Android Developers blog, 35  
OHA, 6, 35  
    <permission> tag, 135  
permissions, 136  
pre-publication checklist, 361  
Preference class, 274  
PreferenceActivity class, 273  
PreferenceCategory class, 274  
PreferenceFragment class, 250  
PreferenceScreen class, 274  
ProgressDialog class, 260  
provider package, 300  
Rapid Application Development (RAD), 346  
RelativeLayout class, 231  
resources, 168  
responsiveness design, 361  
rs:ResEnum app, 168  
runtime changes, handling, 321  
screens  
    compatibility mode, 306, 321  
    support, 130  
SDK  
    download, 52  
    License Agreement, 52  
    upgrades, 41  
seamlessness design, 361  
SearchRecentSuggestions content provider, 300  
security, 136  
Settings content provider, 300  
SharedPreferences interface, 273  
SharedPreferences.Editor interface, 273  
SHOP4APPS, 393  
signing applications, 384  
Snake application sample code, 81  
Soc.io Mall, 393  
software development process, 346  
source code, 6  
Stack Overflow, 6  
storage, 284  
StrictMode class, 373  
Support4Demos sample app, 51  
Support13Demos sample app, 51  
    <supports-gl-texture> tag, 131

TabActivity class, 231  
TabHost class, 231  
TableLayout class, 231  
testing references, 376  
third-party content providers, 299  
TimePickerDialog class, 260  
Uri class, 81  
user interfaces  
    debugging and profiling, 100  
    guidelines, 361  
    layout controls, 94  
UserDictionary content provider, 300  
<uses-configuration> tag, 129  
<uses-feature> tag, 130  
V CAST Apps, 393  
ViewGroup class, 231  
virtual device management, 421  
VoicemailContract class, 292  
waterfall development, 346  
WebViewFragment class, 250  
Windows USB driver, 39  
Wireless Developer Network, 7  
WURFL, 333  
XDA-Developers Android forum, 7  
**WebView classes, implementing, 243-244**  
**WebViewFragment class, 238, 250**  
**weight attribute, 210**  
**weightSum attribute, 210**  
**white-box testing, 369**  
**widget package, 171**  
**windows (Eclipse)**  
    maximizing, 440  
    minimizing, 440  
    open, limiting, 441  
    side by side view, 440  
**Windows USB driver download website, 39**  
**Wireless Application Protocol (WAP), 15-16**  
**Wireless Developer Network website, 7**  
**workflow**  
    flexibility, improving, 111-113  
    screens, 234  
**working squares, 306-307**  
**workspace (Eclipse)**  
    perspectives, 440  
    source control services, 439  
    tabs, closing, 441  
    two sections of same file, viewing, 441  
    windows  
        limiting open, 441  
        maximizing/minimizing, 440  
        side by side view, 440  
**writeable files, 277**  
**writing files, 279**  
**WURFL website, 333**

---

**X-Z**

---

**XDA-Developers Android forum, 7**

**XML**

    escaping, 147  
    files, 159  
        accessing, 160  
        defining, 160  
        parsing utilities, 280  
        reading, 280-281  
        storing, 139  
    layout resource files, creating, 199-200  
    pull parsing package, 45  
    SAX support package, 45

**zipalign tool, 99**

*This page intentionally left blank*