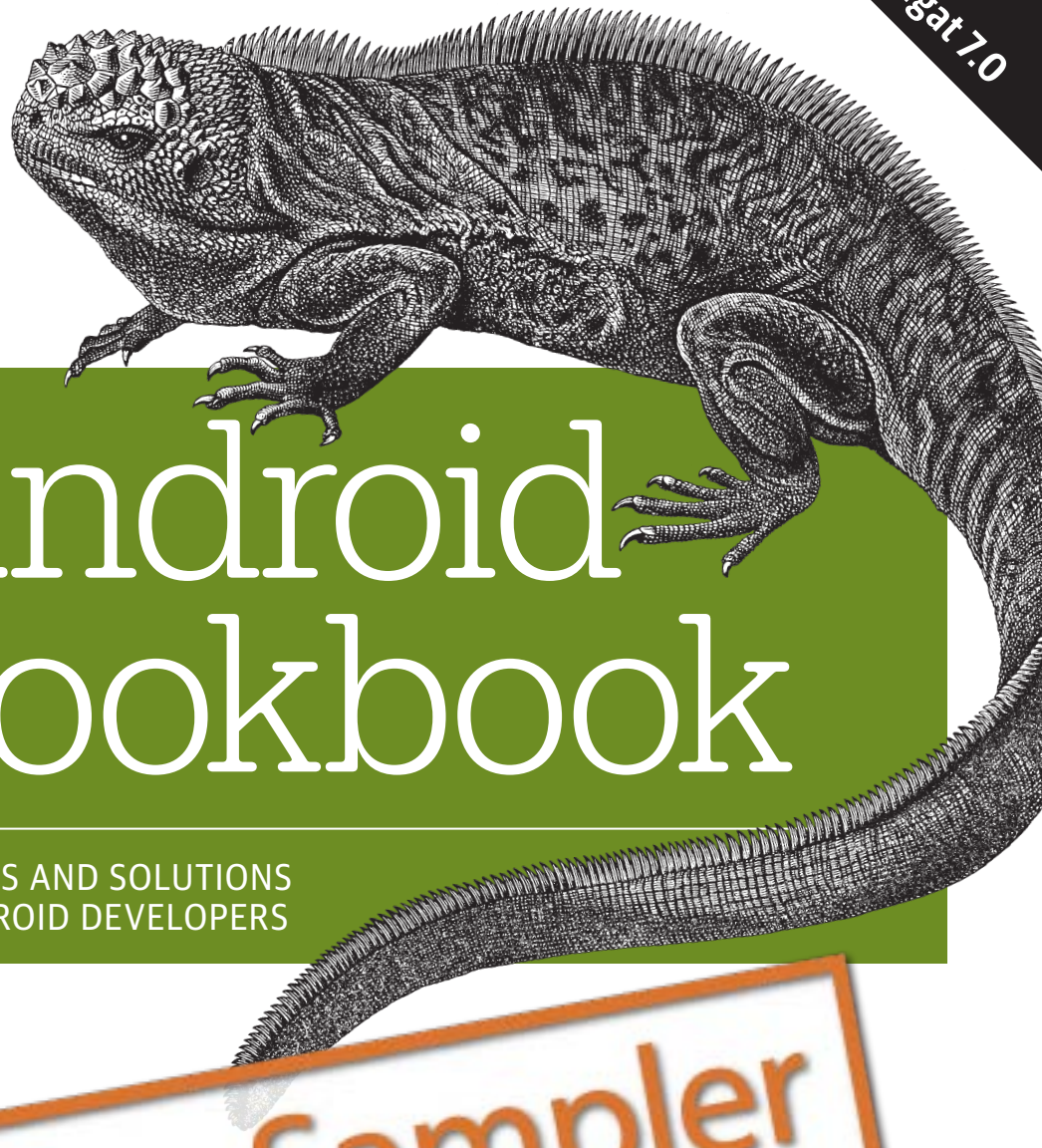
O'REILLY®

2nd Edition
Covers Android Nougat 7.0



Android Cookbook

PROBLEMS AND SOLUTIONS
FOR ANDROID DEVELOPERS

Free Sampler

Ian F. Darwin

O'REILLY®

Android Cookbook

Jump in and build working Android apps with the help of more than 230 tested recipes. The second edition of this acclaimed cookbook includes recipes for working with user interfaces, multitouch gestures, location awareness, web services, and specific device features such as the phone, camera, and accelerometer. You also get useful info on packaging your app for the Google Play Market.

Ideal for developers familiar with Java, Android basics, and the Java SE API, this book features recipes contributed by more than three dozen Android developers. Each recipe provides a clear solution and sample code you can use in your project right away. Among numerous topics, this cookbook helps you:

- Get started with the tooling you need for developing and testing Android apps
- Create layouts with Android's UI controls, graphical services, and pop-up mechanisms
- Build location-aware services on Google Maps and OpenStreetMap
- Control aspects of Android's music, video, and other multimedia capabilities
- Work with accelerometers and other Android sensors
- Use various gaming and animation frameworks
- Store and retrieve persistent data in files and embedded databases
- Access RESTful web services with JSON and other formats
- Test and troubleshoot individual components and your entire application

"This book is here to help the Android developer community share the knowledge that will help make better apps. Those who contribute knowledge here are helping to make Android development easier for those who come after."

—from the Preface

Ian F. Darwin has worked in the computer industry for three decades. He wrote the freeware file command used on Linux and BSD and authored *Checking C Programs with Lint*, *Java Cookbook*, and over 70 articles and courses on C and Unix. In addition to programming and consulting, Ian teaches Java, Android, and related topics for Learning Tree International, one of the world's largest technical training companies.

US $69.99 CAN $92.99
ISBN: 978-1-449-37443-3

Twitter: @oreillymedia
facebook.com/oreilly

Android Cookbook
by Ian F. Darwin

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://www.oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Dawn Schanafelt and Meghan Blanchette
Production Editor: Colleen Lobner
Copyeditor: Kim Cofer
Proofreader: Rachel Head
Indexer: Judith McConville
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

May 2017: Second Edition

Revision History for the Second Edition
2017-05-05: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449374433* for release details.

978-1-449-37443-3

[LSI]

Table of Contents

Preface xiii

1. Getting Started 1
1.1 Understanding the Android Application Architecture 1
1.2 Understanding the Android Activity Life Cycle 3
1.3 Learning About Android Releases 5
1.4 Learning the Java Language 7
1.5 Creating a "Hello, World" Application from the Command Line 8
1.6 Creating a "Hello, World" App with Apache Maven 13
1.7 Choosing an IDE for Android Development 15
1.8 Setting Up Android Studio 18
1.9 Installing Platform Editions and Keeping the SDK Updated 21
1.10 Creating a "Hello, World" App Using Android Studio 25
1.11 Converting an Eclipse ADT Project to Android Studio 30
1.12 Preserving History While Converting from Eclipse to Android Studio 34
1.13 Building an Android Application with both Eclipse and Android Studio 36
1.14 Setting Up Eclipse with AndMore (Replacing ADT) 39
1.15 Creating a "Hello, World" Application Using Eclipse 46
1.16 Installing the Eclipse Marketplace Client in Your Eclipse 51
1.17 Upgrading a Project from Eclipse ADT to Eclipse AndMore 53
1.18 Controlling Emulators/Devices Using Command-Line ADB 57
1.19 Sharing Java Classes from Another Eclipse Project 59
1.20 Referencing Libraries to Implement External Functionality 62
1.21 Using New Features on Old Devices via the Compatibility Libraries 67
1.22 Using SDK Samples to Help Avoid Head Scratching 68
1.23 Taking a Screenshot/Video from the Emulator/Android Device 70
1.24 Program: A Simple CountDownTimer Example 76
1.25 Program: Tipster, a Tip Calculator for the Android OS 79

2. Designing a Successful Application. 97
2.1 Exception Handling 101
2.2 Requesting Android Permissions at Runtime 104
2.3 Accessing Android's Application Object as a "Singleton" 106
2.4 Keeping Data When the User Rotates the Device 109
2.5 Monitoring the Battery Level of an Android Device 111
2.6 Creating Splash Screens in Android 113
2.7 Designing a Conference/Camp/Hackathon/Institution App 117
2.8 Using Google Analytics in an Android Application 119
2.9 Setting First-Run Preferences 122
2.10 Formatting Numbers 123
2.11 Formatting with Correct Plurals 127
2.12 Formatting the Time and Date for Display 130
2.13 Simplifying Date/Time Calculations with the Java 8 java.time API 132
2.14 Controlling Input with KeyListeners 134
2.15 Backing Up Android Application Data 137
2.16 Using Hints Instead of Tool Tips 144

3. Application Testing. 147
3.1 Setting Up an Android Virtual Device (AVD) for App Testing 148
3.2 Testing on a Wide Range of Devices with Cloud-Based Testing 154
3.3 Testing with Eclipse and JUnit 155
3.4 Testing with Android Studio and JUnit 158
3.5 Testing with Robolectric and JUnit 4 163
3.6 Testing with ATSL, Espresso, and JUnit 4 166
3.7 Troubleshooting Application Crashes 170
3.8 Debugging Using Log.d() and LogCat 173
3.9 Getting Bug Reports Automatically with Crash Reporting 175
3.10 Using a Local Runtime Application Log for Analysis of Field Errors or Situations 178
3.11 Reproducing Activity Life-Cycle Scenarios for Testing 181
3.12 Keeping Your App Snappy with StrictMode 186
3.13 Static Code Testing with Android Lint 187
3.14 Dynamic Testing with the Monkey Program 189
3.15 Sending Text Messages and Placing Calls Between AVDs 191

4. Inter-/Intra-Process Communication. 193
4.1 Opening a Web Page, Phone Number, or Anything Else with an Intent 194
4.2 Emailing Text from a View 196
4.3 Sending an Email with Attachments 199
4.4 Pushing String Values Using Intent.putExtra() 201
4.5 Retrieving Data from a Subactivity Back to Your Main Activity 202

4.6 Keeping a Background Service Running While Other Apps Are on Display 205
4.7 Sending/Receiving a Broadcast Message 207
4.8 Starting a Service After Device Reboot 208
4.9 Creating a Responsive Application Using Threads 209
4.10 Using AsyncTask to Do Background Processing 210
4.11 Sending Messages Between Threads Using an Activity Thread Queue and Handler 218
4.12 Creating an Android Epoch HTML/JavaScript Calendar 220

5. Graphics. 227
5.1 Using a Custom Font 227
5.2 Drawing a Spinning Cube with OpenGL ES 230
5.3 Adding Controls to the OpenGL Spinning Cube 234
5.4 Freehand Drawing Smooth Curves 237
5.5 Taking a Picture Using an Intent 242
5.6 Taking a Picture Using android.media.Camera 244
5.7 Scanning a Barcode or QR Code with the Google ZXing Barcode Scanner 248
5.8 Using AndroidPlot to Display Charts and Graphs 251
5.9 Using Inkscape to Create an Android Launcher Icon from OpenClipArt.org 254
5.10 Using Paint.NET to Create Launcher Icons from OpenClipArt.org 259
5.11 Using Nine Patch Files 267
5.12 Creating HTML5 Charts with Android RGraph 270
5.13 Adding a Simple Raster Animation 274
5.14 Using Pinch to Zoom 278

6. Graphical User Interface. 281
6.1 Understanding and Following User Interface Guidelines 282
6.2 Looking Good with Material Design 283
6.3 Choosing a Layout Manager (a.k.a. ViewGroup) and Arranging Components 287
6.4 Handling Configuration Changes by Decoupling the View from the Model 288
6.5 Controlling the Action Bar 291
6.6 Adding a Share Action to Your Action Bar 295
6.7 Building Modern UIs with the Fragment API 299
6.8 Creating a Button and Its Click Event Listener 304
6.9 Enhancing UI Design Using Image Buttons 305
6.10 Using a FloatingActionButton 306
6.11 Wiring Up an Event Listener in Many Different Ways 309
6.12 Using CheckBoxes and RadioButtons 314
6.13 Using Card Widgets 318
6.14 Offering a Drop-Down Chooser via the Spinner Class 320

6.15 Handling Long-Press/Long-Click Events 323
6.16 Displaying Text Fields with TextView and EditText 324
6.17 Constraining EditText Values with Attributes and the TextWatcher Interface 325
6.18 Implementing AutoCompleteTextView 328
6.19 Feeding AutoCompleteTextView Using a SQLite Database Query 330
6.20 Turning Edit Fields into Password Fields 332
6.21 Changing the Enter Key to "Next" on the Soft Keyboard 333
6.22 Processing Key-Press Events in an Activity 336
6.23 Let Them See Stars: Using RatingBar 337
6.24 Making a View Shake 341
6.25 Providing Haptic Feedback 342
6.26 Navigating Different Activities Within a TabView 346
6.27 Creating a Loading Screen that Will Appear Between Two Activities 347
6.28 Adding a Border with Rounded Corners to a Layout 349
6.29 Detecting Gestures in Android 351
6.30 Creating a Simple App Widget 358

7. GUI Alerts: Menus, Dialogs, Toasts, Snackbars, and Notifications. 363
7.1 Alerting the User with Toast and Snackbar 364
7.2 Customizing the Appearance of a Toast 366
7.3 Creating and Displaying a Menu 367
7.4 Handling Choice Selection in a Menu 369
7.5 Creating a Submenu 370
7.6 Creating a Pop-up/Alert Dialog 372
7.7 Using a Timepicker Widget 374
7.8 Creating an iPhone-like WheelPicker for Selection 376
7.9 Creating a Tabbed Dialog 379
7.10 Creating a ProgressDialog 382
7.11 Creating a Custom Dialog with Buttons, Images, and Text 383
7.12 Creating a Reusable "About Box" Class 385
7.13 Creating a Notification in the Status Bar 389

8. Other GUI Elements: Lists and Views. 395
8.1 Building List-Based Applications with RecyclerView 395
8.2 Building List-Based Applications with ListView 399
8.3 Creating a "No Data" View for ListViews 403
8.4 Creating an Advanced ListView with Images and Text 405
8.5 Using Section Headers in ListViews 409
8.6 Keeping the ListView with the User's Focus 413
8.7 Writing a Custom List Adapter 414
8.8 Using a SearchView to Search Through Data in a ListView 418

8.9 Handling Orientation Changes: From ListView Data Values to Landscape Charting 420

9. Multimedia. 425
9.1 Playing a YouTube Video 425
9.2 Capturing Video Using MediaRecorder 426
9.3 Using Android's Face Detection Capability 429
9.4 Playing Audio from a File 432
9.5 Playing Audio Without Interaction 435
9.6 Using Speech to Text 437
9.7 Making the Device Speak with Text-to-Speech 438

10. Data Persistence. 441
10.1 Reading and Writing Files in Internal and External Storage 442
10.2 Getting File and Directory Information 446
10.3 Reading a File Shipped with the App Rather than in the Filesystem 451
10.4 Getting Space Information About the SD Card 453
10.5 Providing a Preference Activity 454
10.6 Checking the Consistency of Default Shared Preferences 459
10.7 Using a SQLite Database in an Android Application 461
10.8 Performing Advanced Text Searches on a SQLite Database 464
10.9 Working with Dates in SQLite 470
10.10 Exposing Non-SQL Data as a SQL Cursor 472
10.11 Displaying Data with a CursorLoader 475
10.12 Parsing JSON Using JSONObject 478
10.13 Parsing an XML Document Using the DOM API 480
10.14 Storing and Retrieving Data via a Content Provider 482
10.15 Writing a Content Provider 483
10.16 Adding a Contact Through the Contacts Content Provider 487
10.17 Reading Contact Data Using a Content Provider 490
10.18 Implementing Drag and Drop 492
10.19 Sharing Files via a FileProvider 496
10.20 Backing Up Your SQLite Data to the Cloud with a SyncAdapter 501
10.21 Storing Data in the Cloud with Google Firebase 510

11. Telephone Applications. 517
11.1 Doing Something When the Phone Rings 517
11.2 Processing Outgoing Phone Calls 521
11.3 Dialing the Phone 525
11.4 Sending Single-part or Multipart SMS Messages 527
11.5 Receiving an SMS Message 529
11.6 Using Emulator Controls to Send SMS Messages to the Emulator 531

11.7 Using Android's TelephonyManager to Obtain Device Information 532

12. Networked Applications. 543
12.1 Consuming a RESTful Web Service Using a URLConnection 544
12.2 Consuming a RESTful Web Service with Volley 547
12.3 Notifying Your App with Google Cloud Messaging "Push Messaging" 549
12.4 Extracting Information from Unstructured Text Using Regular Expressions 558
12.5 Parsing RSS/Atom Feeds Using ROME 559
12.6 Using MD5 to Digest Clear Text 564
12.7 Converting Text into Hyperlinks 565
12.8 Accessing a Web Page Using a WebView 566
12.9 Customizing a WebView 567
12.10 Writing an Inter-Process Communication Service 568

13. Gaming and Animation. 575
13.1 Building an Android Game Using flixel-gdx 576
13.2 Building an Android Game Using AndEngine 580
13.3 Processing Timed Keyboard Input 587

14. Social Networking. 589
14.1 Authenticating Users with OAUTH2 589
14.2 Integrating Social Networking Using HTTP 593
14.3 Loading a User's Twitter Timeline Using HTML or JSON 596

15. Location and Map Applications. 599
15.1 Getting Location Information 599
15.2 Accessing GPS Information in Your Application 601
15.3 Mocking GPS Coordinates on a Device 603
15.4 Using Geocoding and Reverse Geocoding 606
15.5 Getting Ready for Google Maps API V2 Development 607
15.6 Using the Google Maps API V2 612
15.7 Displaying Map Data Using OpenStreetMap 618
15.8 Creating Overlays in OpenStreetMap Maps 621
15.9 Using a Scale on an OpenStreetMap Map 623
15.10 Handling Touch Events on an OpenStreetMap Overlay 624
15.11 Getting Location Updates with OpenStreetMap Maps 627

16. Accelerometer. 631
16.1 Checking for the Presence or Absence of a Sensor 631
16.2 Using the Accelerometer to Detect Shaking 632
16.3 Checking Whether a Device Is Facing Up or Down 636

16.4 Reading the Temperature Sensor 637

17. Bluetooth. 639
17.1 Enabling Bluetooth and Making the Device Discoverable 639
17.2 Connecting to a Bluetooth-Enabled Device 641
17.3 Accepting Connections from a Bluetooth Device 644
17.4 Implementing Bluetooth Device Discovery 645

18. System and Device Control. 647
18.1 Accessing Phone Network/Connectivity Information 647
18.2 Obtaining Information from the Manifest File 648
18.3 Changing Incoming Call Notification to Silent, Vibrate, or Normal 649
18.4 Copying Text and Getting Text from the Clipboard 652
18.5 Using LED-Based Notifications 652
18.6 Making the Device Vibrate 653
18.7 Determining Whether a Given Application Is Running 654

19. All the World's Not Java: Other Programming Languages and Frameworks. 657
19.1 Learning About Cross-Platform Solutions 658
19.2 Running Shell Commands from Your Application 659
19.3 Running Native C/C++ Code with JNI on the NDK 661
19.4 Getting Started with SL4A, the Scripting Layer for Android 667
19.5 Creating Alerts in SL4A 669
19.6 Fetching Your Google Documents and Displaying Them in a ListView Using SL4A 673
19.7 Sharing SL4A Scripts in QR Codes 676
19.8 Using Native Handset Functionality from a WebView via JavaScript 678
19.9 Building a Cross-Platform App with Xamarin 680
19.10 Creating a Cross-Platform App Using PhoneGap/Cordova 685

20. All the World's Not English: Strings and Internationalization. 689
20.1 Internationalizing Application Text 690
20.2 Finding and Translating Strings 693
20.3 Handling the Nuances of strings.xml 695

21. Packaging, Deploying, and Distributing/Selling Your App. 701
21.1 Creating a Signing Certificate and Using It to Sign Your Application 701
21.2 Distributing Your Application via the Google Play Store 705
21.3 Distributing Your Application via Other App Stores 707
21.4 Monetizing Your App with AdMob 708
21.5 Obfuscating and Optimizing with ProGuard 714
21.6 Hosting Your App on Your Own Server 717

21.7 Creating a “Self-Updating” App 718
21.8 Providing a Link to Other Published Apps in the Google Play Store 720

Index **725**

CHAPTER 2

Designing a Successful Application

This chapter is about design guidelines for writing imaginative and useful Android applications. Several recipes describe specific aspects of successful design. This section will list some others.

One purpose of this chapter is to explain the benefits of developing native Java Android applications over other methods of delivering rich content on mobile devices.

Requirements of a native handset application

There are a number of key requirements for successfully delivering any mobile handset application, regardless of the platform onto which it will be deployed:

- The application should be easy to install, remove, and update on a device.
- It should address the user's needs in a compelling, unique, and elegant way.
- It should be feature-rich while remaining usable by both novice and expert users.
- It should be familiar to users who have accessed the same information through other routes, such as a website.
- Key areas of functionality should be readily accessible.
- It should have a common look and feel with other native applications on the handset, conforming to the target platform's standards and style guidelines.
- It should be stable, scalable, usable, and responsive.
- It should use the platform's capabilities tastefully, when it makes the user's experience more compelling.

Android application design

Colin Wilcox

The Android application we will design in this chapter will exploit the features and functions unique to the Android OS platform. In general, the application will be an Activity-based solution allowing independent and controlled access to data on a screen-by-screen basis. This approach helps to localize potential errors and allows sections of the flow to be readily replaced or enhanced independently of the rest of the application.

Navigation will use a similar approach to that of the Apple iPhone solution in that all key areas of functionality will be accessed from a single navigation bar control. The navigation bar will be accessible from anywhere within the application, allowing the user to freely move around the application.

The Android solution will exploit features inherent to Android devices, supporting the devices' touch-screen features, the hardware button that allows users to switch the application to the background, and application switching capability.

Android provides the ability to jump back into an application at the point where it was switched out. This will be supported, when possible, within this design.

The application will use only standard Android user interface controls to make it as portable as possible. The use of themes or custom controls is outside the scope of this chapter.

The application will be designed such that it interfaces to a thin layer of RESTful web services that provide data in a JSON format. This interface will be the same as the one used by the Apple iPhone, as well as applications written for other platforms.

The application will adopt the Android style and design guidelines wherever possible so that it fits in with other Android applications on the device.

Data that is local to each view will be saved when the view is exited and automatically restored with the corresponding user interface controls repopulated when the view is next loaded.

A number of important device characteristics should be considered, as discussed in the following subsections.

Screen size and density. In order to categorize devices by their screen type, Android defines two characteristics for each device: screen size (the physical dimensions of the screen) and screen density (the physical density of the pixels on the screen, or dpi [dots per inch]). To simplify all the different types of screen configurations, the Android system generalizes them into select groups that make them easier to target.

The designer should take into account the most appropriate choices for screen size and screen density when designing the application.

By default, an application is compatible with all screen sizes and densities, because the Android system makes the appropriate adjustments to the UI layout and image resources. However, you should create specialized layouts for certain screen sizes and provide specialized images for certain densities, by using alternative layout resources and by declaring in your manifest exactly which screen sizes your application supports.

Input configurations. Many devices provide a different type of user input mechanism, such as a hardware keyboard, a trackball, or a five-way navigation pad. If your application requires a particular kind of input hardware, you must declare it in the *AndroidManifest.xml* file, and be aware that the Google Play Store will not display your app on devices that lack this feature. However, it is rare for an application to require a certain input configuration.

Device features. There are many hardware and software features that may or may not exist on a given Android-powered device, such as a camera, a light sensor, Bluetooth capability, a certain version of OpenGL, or the fidelity of the touch screen. You should never assume that a certain feature is available on all Android-powered devices (other than the availability of the standard Android library).

A sophisticated Android application will use both types of menus provided by the Android framework, depending on the circumstances:

- *Options menus* contain primary functionality that applies globally to the current Activity or starts a related Activity. An options menu is typically invoked by a user pressing a hard button, often labeled Menu, or a soft menu button on an Action Bar (a vertical stack of three dots).
- *Context menus* contain secondary functionality for the currently selected item. A context menu is typically invoked by a user performing a long-press (press and hold) on an item. Like on the options menu, the selected operation can run in either the current or another Activity. A context menu is for any commands that apply to the current selection.

The commands on the context menu that appear when you long-press on an item should be duplicated in the Activity you get to by a normal press on that item.

As very general guidance:

- Place the most frequently used operations first in the menu.

- Only the most important commands should appear as buttons on the screen; delegate the rest to the menu.
- Consider moving menu items to the action bar if your application uses one.

The system will automatically lay out the menus and provide standard ways for users to access them, ensuring that the application will conform to the Android user interface guidelines. In this sense, menus are familiar and dependable ways for users to access functionality across all applications.

Our Android application will make extensive use of Google's Intent mechanism for passing data between `Activity` objects. Intents not only are used to pass data between views within a single application, but also allow data, or requests, to be passed to external modules. As such, much functionality can be adopted by the Android application by embedded functionality from other applications invoked by `Intent` calls. This reduces the development process and maintains the common look and feel and functionality behavior across all applications.

Data feeds and feed formats. It is not a good idea to interface directly to any third-party data source; for example, it would be a bad idea to use a Type 3 JDBC driver in your mobile application to talk directly to a database on your server. The normal approach would be to mediate the data, from several sources in potentially multiple data formats, through middleware, which then passes data to an application through a series of RESTful web service APIs in the form of JSON data streams.

Typically, data is provided in such formats as XML, SOAP, or some other XML-derived representation. Representations such as SOAP are heavyweight, and as such, transferring data from the backend servers in this format increases development time significantly as the responsibility of converting this data into something more manageable falls on either the handset application or an object on the middleware server.

Mitigating the source data through a middleware server also helps to break the dependency between the application and the data. Such a dependency has the disadvantage that if, for some reason, the nature of the data changes or the data cannot be retrieved, the application may be broken and become unusable, and such changes may require the application to be republished. Mitigating the data on a middleware server ensures that the application will continue to work, albeit possibly in a limited fashion, regardless of whether the source data exists. The link between the application and the mitigated data will remain.

2.1 Exception Handling

Ian Darwin

Problem

Java has a well-defined exception handling mechanism, but it takes some time to learn to use it effectively without frustrating either users or tech support people.

Solution

Java offers an exception hierarchy that provides considerable flexibility when used correctly. Android offers several mechanisms, including dialogs and toasts, for notifying the user of error conditions. The Android developer should become acquainted with these mechanisms and learn to use them effectively.

Discussion

Java has had two categories of exceptions (actually of `Exception`'s parent, `Throwable`) since it was introduced: checked and unchecked. In Java Standard Edition, apparently the intention was to force the programmer to face the fact that, while certain things could be detected at compile time, others could not. For example, if you were installing a desktop application on a large number of PCs, it's likely that the disk on some of those PCs would be near capacity, and trying to save data on them could fail; meanwhile, on other PCs some file that the application depended upon might have gone missing, not due to programmer error but to user error, filesystem happenstance, gerbils chewing on the cables, or whatever. So the category of `IOException` was created as a "checked exception," meaning that the programmer would have to check for it, either by having a `try-catch` clause inside the file-using method or by having a `throws` clause on the method definition. The general rule, which all well-trained Java developers memorize, is the following:

> `Throwable` is the root of the throwable hierarchy. `Exception`, and all of its subclasses other than `RuntimeException` or any subclass thereof, is checked. All else is unchecked.

This means that `Error` and all of its subclasses are unchecked (see Figure 2-1). If you get a `VMError`, for example, it means there's a bug in the runtime. There's nothing you can do about this as an application programmer. `RuntimeException` subclasses include things like the excessively long-named `ArrayIndexOutOfBoundsException`; this and friends are unchecked because it is your responsibility to catch these exceptions at development time, by testing for them (see Chapter 3).

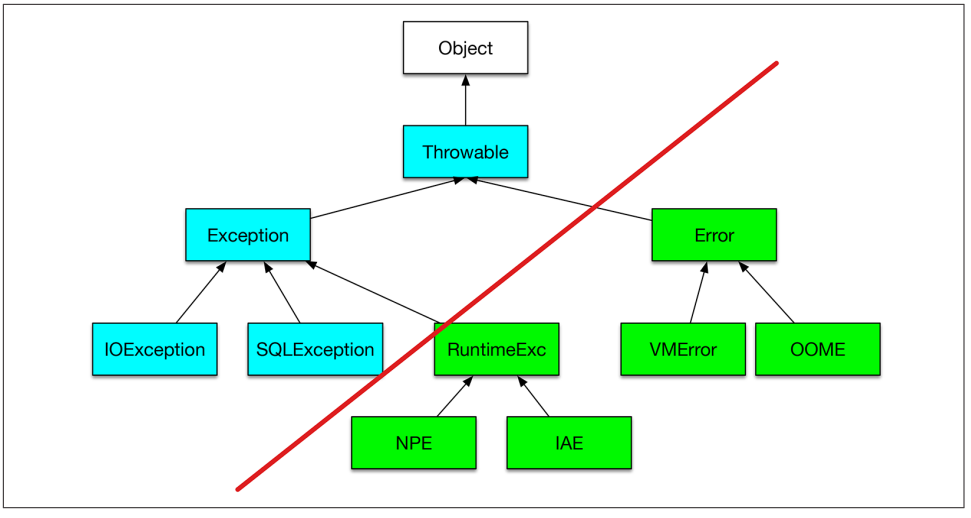


Figure 2-1. Throwable hierarchy

Where to catch exceptions

The (over)use of checked exceptions led a lot of early Java developers to write code that was sprinkled with `try-catch` blocks, partly because the use of the `throws` clause was not emphasized early enough in some training programs and books. As Java itself has moved more to enterprise work, and newer frameworks such as Spring, Hibernate, and JPA have come along and are emphasizing the use of unchecked exceptions, this early position has shifted. It is now generally accepted that you want to catch exceptions as close to the user as possible. Code that is meant for reuse—in libraries or even in multiple applications—should not try to do error handling. What it can do is what's called *exception translation*; that is, turning a technology-specific (and usually checked) exception into a generic, unchecked exception. Example 2-1 shows the basic pattern.

Example 2-1. Exception translation

```
public class ExceptionTranslation {
    public String readTheFile(String f) {
        try (BufferedReader is = new BufferedReader(new FileReader(f))) {
            String line = is.readLine();
            return line;
        } catch (FileNotFoundException fnf) {
            throw new RuntimeException("Could not open file " + f, fnf);
        } catch (IOException ex) {
            throw new RuntimeException("Problem reading file " + f, ex);
        }
    }
}
```

Note that prior to Java 7, you'd have had to write an explicit `finally` clause to close the file:

```
        } finally {
            if (is != null) {
                try {
                    is.close();
                } catch(IOException grr) {
                    throw new RuntimeException("Error on close of " + f, grr);
                }
            }
        }
    }
```

Note how the use of checked exceptions clutters even that code: it is virtually impossible for the `is.close()` to fail, but since you want to have it in a `finally` block (to ensure that it gets tried if the file was opened but then something went wrong), you have to have an additional `try-catch` around it. So, checked exceptions are (more often than not) an annoyance, should be avoided in new APIs, and should be paved over with unchecked exceptions when using code that requires them.

There is an opposing view, espoused by the official Oracle website and others. In a comment on the website from which this book was produced, reader Al Sutton points out the following:

> Checked exceptions exist to force developers to acknowledge that an error condition can occur and that they have thought about how they want to deal with it. In many cases there may be little that can be done beyond logging and recovery, but it is still an acknowledgment by the developer that they have considered what should happen with this type of error. The example shown … stops callers of the method from differentiating between when a file doesn't exist (and thus may need to be re-fetched), and when there is a problem reading the file (and thus the file exists but is unreadable), which are two different types of error conditions.

Android, wishing to be faithful to the Java API, has a number of these checked exceptions (including the ones shown in the example), so they should be treated the same way.

What to do with exceptions

Exceptions should almost always be reported. When I see code that catches exceptions and does nothing at all about them, I despair. They should, however, be reported only once (do not both log and translate/rethrow!). The point of all normal exceptions is to indicate, as the name implies, an exceptional condition. Since on an Android device there is no system administrator or console operator, exceptional conditions need to be reported to the user.

You should think about whether to report exceptions via a dialog or a toast. The exception handling situation on a mobile device is different from that on a desktop

computer. The user may be driving a car or operating other machinery, interacting with people, and so on, so you should not assume you have her full attention.

I know that most examples, even in this book, use a toast, because it involves less coding than a dialog. But remember that a toast will only appear on the screen for a few seconds; blink and you may miss it. If the user needs to do something to correct the problem, you should use a dialog.

Toasts simply pop up and then obliviate. Dialogs require the user to acknowledge an exceptional condition, and either do, or give the app permission to do, something that might cost money (such as turning on internet access in order to run an application that needs to download map tiles).

Use toasts to "pop up" unimportant information; use dialogs to display important information and to obtain confirmation.

See Also

Recipe 3.9.

2.2 Requesting Android Permissions at Runtime

Mike Way

Problem

In Android 6 and later, you must check permissions at runtime in addition to specifying them in the manifest.

Solution

"Dangerous" resources are those that could affect the user's stored information, or privacy, etc. To access resources protected by "dangerous" permissions you must:

- Check if the user has already granted permission before accessing a resource.
- Explicitly request permissions from the user if the permissions have not previously been granted.
- Have an alternate course of action so the application does not crash if the permission is not granted.

Discussion

Before accessing a resource that requires permission, you must first check if the user has already granted permission. To do this, call the `Activity` method `checkSelfPermission(permission)`. It will return either `PERMISSION_GRANTED` or `PERMISSION_DENIED`:

```
if (ActivityCompat.checkSelfPermission(this,
    Manifest.permission.WRITE_EXTERNAL_STORAGE) ==
        PackageManager.PERMISSION_GRANTED) {
    // If you get here then you have the permission and can do some work
} else {
    // See below
}
```

If the preceding check indicates that the permission has not been granted, you must explicitly request it by calling the `Activity` method `requestPermissions()`:

```
void requestPermissions (String[] permissions, int requestCode)
```

As this will interact with the user, it is an asynchronous request. You must override the `Activity` method `onRequestPermissionsResult()` to get notified of the response:

```
public void onRequestPermissionsResult(
    int requestCode, String[] permissions, int[] grantResults);
```

For example:

```
// Unique request code for the particular permissions request
private static int REQUEST_EXTERNAL_STORAGE = 1;
...
// Request the permission from the user
ActivityCompat.requestPermissions(this,
    new String[]{Manifest.permission.WRITE_EXTERNAL_STORAGE },
    REQUEST_EXTERNAL_STORAGE);

// Callback handler for the eventual response
@Override
public void onRequestPermissionsResult(
    int requestCode, String[] permissions, int[] grantResults) {

  boolean granted = true;
  if (requestCode == REQUEST_EXTERNAL_STORAGE) {
          // Received permission result for external storage permission.
           Log.i(TAG, "Got response for external storage permission request.");

              // Check if all the permissions have been granted
              if (grantResults.length > 0 ) {
                  for (int result : grantResults) {
                      if (result != PackageManager.PERMISSION_GRANTED) {
                          granted = false;
                      }
                  }
              } else {
                  granted = false;
```

```
            }
        }
        ...
        // If granted is true: carry on and perform the action. Calling
        // checkSelfPermission() will now return PackageManager.PERMISSION_GRANTED
```

It is usually a good idea to provide the user with information as to why the permissions are required. To do this you call the `Activity` method `boolean shouldShowRequestPermissionRationale(String permission)`. If the user has previously refused to grant the permissions this method will return `true`, giving you the opportunity to display extra information as to why they should be granted:

```
if (ActivityCompat.shouldShowRequestPermissionRationale(this,
        Manifest.permission.WRITE_EXTERNAL_STORAGE)) {
    // Provide additional info if the permission was not granted
    // and the user would benefit from additional
    // context for the use of the permission
    Log.i(TAG, "Displaying permission rationale to provide additional context.");
    Snackbar.make(mLayout, R.string.external_storage_rationale,
            Snackbar.LENGTH_INDEFINITE)
            .setAction(R.string.ok, new View.OnClickListener() {
                @Override
                public void onClick(View view) {
                        ActivityCompat.requestPermissions(MainActivity.this,
                            new String[]{
                                    Manifest.permission.WRITE_EXTERNAL_STORAGE,
                                    Manifest.permission.READ_EXTERNAL_STORAGE},
                            REQUEST_EXTERNAL_STORAGE);
                }
            }).show();
```

This uses a `Snackbar` (see Recipe 7.1) to display the rationale, until the user clicks the `Snackbar` to dismiss it.

See Also

This permission checking technique is also used in the example project in Recipe 14.1. There is more documention at the official documentation site.

Source Download URL

The source code for this example is in the Android Cookbook repository, in the subdirectory *PermissionRequest* (see "Getting and Using the Code Examples" on page 18).

2.3 Accessing Android's Application Object as a "Singleton"

Adrian Cowham

Problem

You need to access "global" data from within your Android app.

Solution

The best solution is to subclass `android.app.Application` and treat it as a singleton with static accessors. Every Android app is guaranteed to have exactly one `android.app.Application` instance for the lifetime of the app. If you choose to subclass `android.app.Application`, Android will create an instance of your class and invoke the `android.app.Application` life-cycle methods on it. Because there's nothing preventing you from creating another instance of your subclassed `android.app.Application`, it isn't a genuine singleton, but it's close enough.

Having global access to such objects as session handlers, web service gateways, or anything that your application only needs a single instance of will dramatically simplify your code. Sometimes these objects can be implemented as singletons, and sometimes they cannot because they require a `Context` instance for proper initialization. In either case, it's still valuable to add static accessors to your subclassed `android.app.Application` instance so that you can consolidate all globally accessible data in one place, have guaranteed access to a `Context` instance, and easily write "correct" singleton code without having to worry about synchronization.

Discussion

When writing your Android app you may find it necessary to share data and services across multiple Activities. For example, if your app has session data, such as the identity of the currently logged-in user, you will likely want to expose this information. When developing on the Android platform, the pattern for solving this problem is to have your `android.app.Application` instance own all global data, and then treat your `Application` instance as a singleton with static accessors to the various data and services.

When writing an Android app you're guaranteed to only have one instance of the `android.app.Application` class, so it's safe (and recommended by the Google Android team) to treat it as a singleton. That is, you can safely add a static `getInstance()` method to your `Application` implementation. Example 2-2 provides an example.

Example 2-2. The Application implementation

```
public class AndroidApplication extends Application {

    private static AndroidApplication sInstance;

    private SessionHandler sessionHandler; // Generic your-application handler
```

```
    public static AndroidApplication getInstance() {
      return sInstance;
    }

    public Session Handler getSessionHandler()
        return sessionHandler;
    }

    @Override
    public void onCreate() {
      super.onCreate();
      sInstance = this;
      sInstance.initializeInstance();
    }

    protected void initializeInstance() {
        // Do all your initialization here
        sessionHandler = new SessionHandler(
            this.getSharedPreferences( "PREFS_PRIVATE", Context.MODE_PRIVATE ) );
    }

    /** This is a stand-in for some application-specific session handler;
     * would normally be a self-contained public class.
     */
    private class SessionHandler {
        SharedPreferences sp;
        SessionHandler(SharedPreferences sp) {
            this.sp = sp;
        }
    }
}
```

This isn't the classical singleton implementation, but given the constraints of the Android framework it's the closest thing we have; it's safe, and it works.

The notion of the "session handler" is that it keeps track of per-user information such as name and perhaps password, or any other relevant information, across different Activities and the same Activity even if it gets destroyed and re-created. Our `SessionHandler` class is a placeholder for you to compose such a handler, using whatever information you need to maintain across Activities!

Using this technique in this app has simplified and cleaned up the implementation. Also, it has made it much easier to develop tests. Using this technique in conjunction with the Robolectric testing framework (see Recipe 3.5), you can mock out the entire execution environment in a straightforward fashion.

Also, don't forget to add the application class's `android:"name"` declaration to the existing `application` element in your *AndroidManifest.xml* file:

```
<application android:icon="@drawable/app_icon"
  android:label="@string/app_name"
  android:name=".AndroidApplication">
```

See Also

My blog post.

Source Download URL

The source code for this project is in the Android Cookbook repository, in the subdirectory *AppSingleton* (see "Getting and Using the Code Examples" on page 18).

2.4 Keeping Data When the User Rotates the Device

Ian Darwin

Problem

When the user rotates the device, Android will normally destroy and re-create the current Activity. You want to keep some data across this cycle, but all the fields in your Activity are lost during it.

Solution

There are several approaches. If all your data comprises primitive types, consists of `Strings`, or is `Serializable`, you can save it in `onSaveInstanceState()` in the `Bundle` that is passed in.

Another solution lets you return a single arbitrary object. You need only override `onRetainNonConfigurationInstance()` in your Activity to save some values, call `getLastNonConfigurationInstance()` near the end of your `onCreate()` method to see if there is a previously saved value, and, if so, assign your fields accordingly.

Discussion

Using onSaveInstanceState()

See Recipe 1.2.

Using onRetainNonConfigurationInstance()

The `getLastNonConfigurationInstance()` method's return type is `Object`, so you can return any value you want from it. You might want to create a `Map` or write an inner class in which to store the values, but it's often easier just to pass a reference to the current Activity, for example, using `this`:

```
public class MyActivity extends Activity {
                ...

        /** Returns arbitrary single token object to keep alive across
```

```
     * the destruction and re-creation of the entire Enterprise.
     */
    @Override
    public Object onRetainNonConfigurationInstance() {
            return this;
    }
```

The preceding method will be called when Android destroys your main Activity. Suppose you wanted to keep a reference to another object that was being updated by a running service, which is referred to by a field in your Activity. There might also be a Boolean to indicate whether the service is active. In the preceding code, we return a reference to the Activity from which all of its fields can be accessed (even private fields, since the outgoing and incoming `Activity` objects are of the same class). In my geotracking app JPSTrack, for example, I have a `FileSaver` class that accepts data from the location service; I want it to keep getting the location, and saving it to disk, in spite of rotations, rather than having to restart it every time the screen rotates. Rotation is unlikely if the device is anchored in a car dash mount (we hope), but quite likely if a passenger, or a pedestrian, is taking pictures or typing notes while geotracking.

After Android creates the new instance, it calls `onCreate()` to notify the new instance that it has been created. In `onCreate()` you typically do constructor-like actions such as initializing fields and assigning event listeners. You still need to do those, so leave them alone. Near the end of `onCreate()`, however, you will add some code to get the old instance, if there is one, and get some of the important fields from it. The code should look something like Example 2-3.

Example 2-3. The onCreate() method

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    saving = false;
    paused = false;

    // Lots of other initializations...

    // Now see if we just got interrupted by, e.g., rotation
        Main old = (Main) getLastNonConfigurationInstance();
        if (old != null) {
            saving = old.saving;
            paused = old.paused;

            // This is the most important line: keep saving to same file!
            fileSaver = old.fileSaver;
            if (saving) {
                    fileNameLabel.setText(fileSaver.getFileName());
            }
```

```
            return;
            }

            // I/O helper
            fileSaver = new GPSFileSaver(...);
}
```

The `fileSaver` object is the big one, the one we want to keep running and not re-create every time. If we don't have an old instance, we create the `fileSaver` only at the very end of `onCreate()`, since otherwise we'd be creating a new one just to replace it with the old one, which is (at the least) bad for performance. When the `onCreate()` method finishes, we hold no reference to the old instance, so it should be eligible for Java garbage collection. The net result is that the Activity appears to keep running nicely across screen rotations, despite the re-creation.

An alternative possibility is to set `android:configChanges="orientation"` in your *AndroidManifest.xml*. This approach prevents the Activity from being destroyed and re-created, but typically also prevents the application from displaying correctly in landscape mode, and is officially regarded as not good practice—see the following reference.

See Also

Recipe 2.3, the developer documentation on handling configuration changes.

Source Download URL

You can download the source code for this example from GitHub. Note that if you want it to compile, you will also need the jpstrack project, from the same GitHub account.

2.5 Monitoring the Battery Level of an Android Device

Pratik Rupwal

Problem

You want to detect the battery level on an Android device so that you can notify the user when the battery level goes below a certain threshold, thereby avoiding unexpected surprises.

Solution

A broadcast receiver that receives the broadcast message sent when the battery status changes can identify the battery level and can issue alerts to users.

Discussion

Sometimes we need to show an alert to the user when the battery level of an Android device goes below a certain limit. The code in Example 2-4 sets the broadcast message to be sent whenever the battery level changes and creates a broadcast receiver to receive the broadcast message, which can alert the user when the battery gets discharged below a certain level.

Example 2-4. The main Activity

```
public class MainActivity extends Activity {

 /** Called when the Activity is first created. */
 @Override
 public void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     setContentView(R.layout.main);

    /** This registers the receiver for a broadcast message to be sent
    * to when the battery level is changed. */

     this.registerReceiver(this.myBatteryReceiver,
       new IntentFilter(Intent.ACTION_BATTERY_CHANGED));

     /** Intent.ACTION_BATTERY_CHANGED can be replaced with
      * Intent.ACTION_BATTERY_LOW for receiving
      * a message only when battery level is low rather than sending
      * a broadcast message every time battery level changes.
      * There is also ACTION_BATTERY_OK for when the battery
      * has been charged a certain amount above the level that
      * would trigger the low condition.
      */
 }

 private BroadcastReceiver myBatteryReceiver =
    new BroadcastReceiver() {

   @Override
   public void onReceive(Context ctx, Intent intent) {
    // bLevel is battery percent-full as an integer
    int bLevel = intent.getIntExtra("level", 0);
    Log.i("BatteryMon", "Level now " + bLevel);
   }
 };
}
```

The `ACTION_BATTERY_LOW` and `ACTION_BATTERY_OK` levels are not documented, and are settable only by rebuilding the operating system, but they may be around 10 and 15, or 15 and 20, respectively.

2.6 Creating Splash Screens in Android

Rachee Singh and Ian Darwin

Problem

You want to create a splash screen that will appear while an application is loading.

Solution

You can construct a splash screen as an Activity or as a dialog. Since its purpose is accomplished within a few seconds, it can be dismissed after a short time interval has elapsed or upon the click of a button in the splash screen.

Discussion

The splash screen was invented in the PC era, initially as a cover-up for slow GUI construction when PCs were slow. Vendors have kept them for branding purposes. But in the mobile world, where the longest app start-up time is probably less than a second, people are starting to recognize that splash screens have become somewhat anachronistic. When I (Ian Darwin) worked at eHealth Innovation, we recognized this by making the splash screen for our BANT application disappear after just one second. The question arises whether we still need splash screens at all. With most mobile apps, the name and logo appear in the app launcher, and on lots of other screens within the app. Is it time to make the splash screen disappear altogether?

The answer to that question is left up to you and your organization. For completeness, here are two methods of handling the application splash screen.

The first version uses an Activity that is dedicated to displaying the splash screen. The splash screen displays for two seconds or until the user presses the Menu key, and then the main Activity of the application appears. First we use a thread to wait for a fixed number of seconds, and then we use an Intent to start the real main Activity. The one downside to this method is that your "main" Activity in your *AndroidManifest.xml* file is the splash Activity, not your real main Activity. Example 2-5 shows the splash Activity.

Example 2-5. The splash Activity

```
public class SplashScreen extends Activity {
    private long ms=0;
    private long splashTime=2000;
    private boolean splashActive = true;
    private boolean paused=false;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.splash);
        Thread mythread = new Thread() {
            public void run() {
                try {
                    while (splashActive && ms < splashTime) {
                        if(!paused)
                            ms=ms+100;
                        sleep(100);
                    }
                } catch(Exception e) {}
                finally {
                    Intent intent = new Intent(SplashScreen.this, Main.class);
                    startActivity(intent);
                }
            }
        };
        mythread.start();
    }

}
```

Example 2-6 shows the layout of the splash Activity, *splash.xml*.

Example 2-6. The splash layout

```
<?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">

    <ImageView android:src="@drawable/background"
    android:id="@+id/image"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

    <ProgressBar android:id="@+id/progressBar1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal">
    </ProgressBar>
</LinearLayout>
```

One additional requirement is to put the attribute `android:noHistory="true"` on the splash Activity in your *AndroidManifest.xml* file so that this Activity will not appear in the history stack, meaning if the user uses the Back button from the main app he will go to the expected Home screen, not back into your splash screen (see Figure 2-2).

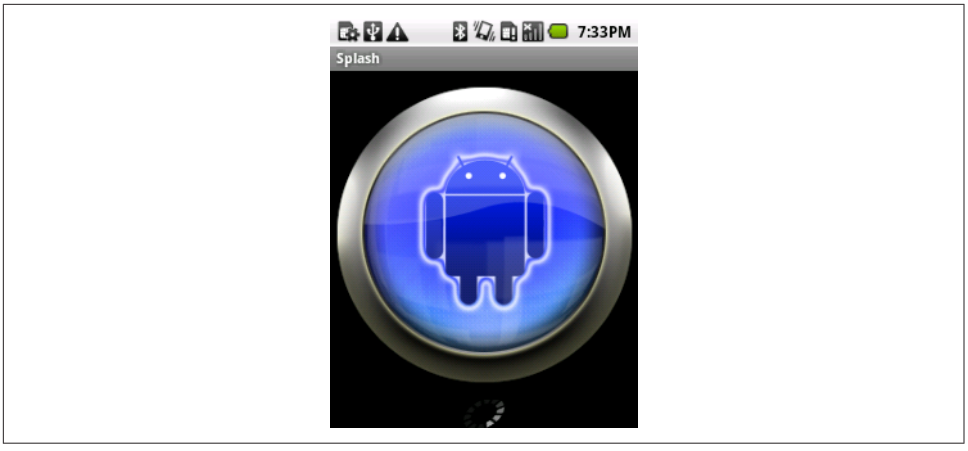


Figure 2-2. Splash screen

Two seconds later, this Activity leads to the next Activity, which is the standard "Hello, World" Android Activity, as a proxy for your application's main Activity (see Figure 2-3).

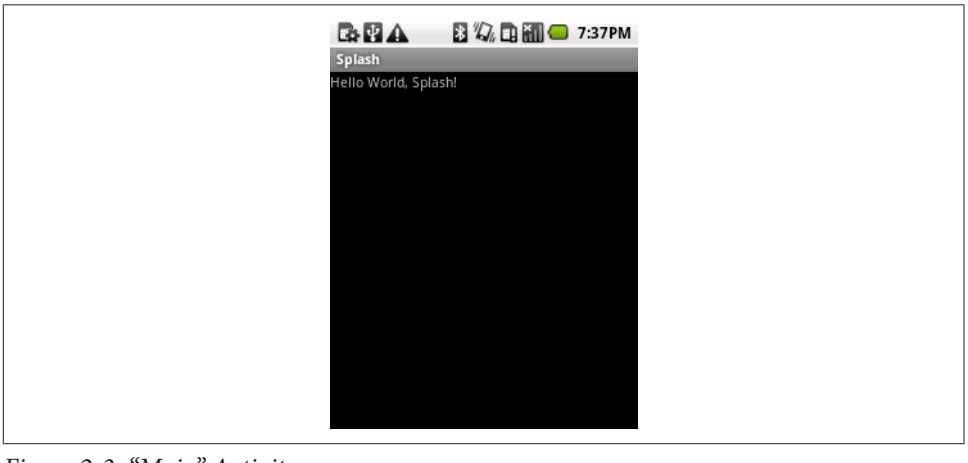


Figure 2-3. "Main" Activity

In the second version (Example 2-7), the splash screen displays until the Menu key on the Android device is pressed, then the main Activity of the application appears. For this, we add a Java class that displays the splash screen. We check for the pressing of the Menu key by checking the `KeyCode` and then finishing the Activity (see Example 2-7).

Example 2-7. Watching for KeyCodes

```
public class SplashScreen extends Activity {
    private long ms=0;
    private long splashTime=2000;
    private boolean splashActive = true;
    private boolean paused=false;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.splash);
    }

    public boolean onKeyDown(int keyCode, KeyEvent event) {
        super.onKeyDown(keyCode, event);
        if (KeyEvent.KEYCODE_MENU == keyCode) {
            Intent intent = new Intent(SplashScreen.this, Main.class);
            startActivity(intent);
        }
        if (KeyEvent.KEYCODE_BACK == keyCode) {
            finish();
        }
        return false;
    }
}
```

The layout of the splash Activity, *splash.xml*, is unchanged from the earlier version.

As before, after the button press this Activity leads to the next Activity, which represents the main Activity.

The other major method involves use of a dialog, started from the `onCreate()` method in your main method. This has a number of advantages, including a simpler Activity stack and the fact that you don't need an extra Activity that's only used for the first few seconds. The disadvantage is that it takes a bit more code, as you can see in Example 2-8.

Example 2-8. The splash dialog

```
public class SplashDialog extends Activity {
    private Dialog splashDialog;

    /** Called when the Activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        StateSaver data = (StateSaver) getLastNonConfigurationInstance();
        if (data != null) { // "All this has happened before"
            if (data.showSplashScreen ) { // And we didn't already finish
                showSplashScreen();
            }
            setContentView(R.layout.main);
```

```
            // Do any UI rebuilding here using saved state
        } else {
            showSplashScreen();
            setContentView(R.layout.main);
            // Start any heavy-duty loading here, but on its own thread
        }
    }
```

The basic idea is to display the splash dialog at application startup, but also to redisplay it if you get, for example, an orientation change while the splash screen is running, and to be careful to remove it at the correct time if the user backs out or if the timer expires while the splash screen is running.

See Also

Ian Clifton's blog post titled "Android Splash Screens Done Right" argues passionately for the dialog method.

Source Download URL

The source code for this example is in the Android Cookbook repository, in the subdirectory *SplashDialog* (see "Getting and Using the Code Examples" on page 18).

2.7 Designing a Conference/Camp/Hackathon/Institution App

Ian Darwin

Problem

You want to design an app for use at a conference, BarCamp, or hackathon, or inside a large institution such as a hospital.

Solution

Provide at least the required functions listed in this recipe's "Discussion" section, and as many of the optional ones as you think make sense.

Discussion

A good app of this type requires some or most of the following functions, as appropriate:

- A map of the building, showing the locations of meetings, food services, washrooms, emergency exits, and so on. You get extra points if you provide a visual slider for moving up or down levels if your conference takes place on more than

one floor or level in the building (think about a 3D fly-through of San Francisco's Moscone Center, including the huge escalators). Remember that some people may know the building, but others will not. Consider having a "where am I" function (the user will type in the name or number of a room he sees; you get extra points if you offer visual matching or use the GPS instead of making the user type) as well as a "where is" function (the user selects from a list and the application jumps to the map view with a pushpin showing the desired location). Turn-by-turn walking directions through a maze of twisty little passages?

- A map of the exhibit hall (if there is a show floor, have a map and an easy way to find a given exhibitor). Ditto for poster papers if your conference features these.
- A schedule view. Highlight changes in red as they happen, including additions, last-minute cancellations, and room changes.
- A sign-up button if your conference has Birds of a Feather (BOF) gatherings; you might even want a "Suggest a new BOF" Activity.
- A local area map. This could be OpenStreetMap or Google Maps, or maybe something more detailed than the standard map. Add folklore, points of interest, navigation shortcuts, and other features. Limit it to a few blocks so that you can get the details right. A university campus is about the right size.
- An overview map of the city. Again, this is not the Google map, but an artistic, neighborhood/zone view with just the highlights.
- Tourist attractions within an hour of the site. Your mileage may vary.
- A food finder. People always get tired of convention food and set out on foot to find something better to eat.
- A friend finder. If Google's Latitude app were open to use by third-party apps, you could tie into Google's data. If it's a security conference, implement this functionality yourself.
- Private voice chat. If it's a small security gathering, provide a Session Initiation Protocol (SIP) server on a well-connected host, with carefully controlled access; it should be possible to have almost walkie talkie–like service.
- Sign-ups for impromptu group formation for trips to tourist attractions or any other purpose.
- Functionality to post comments to Twitter, Facebook, and LinkedIn.
- Note taking! Many people will have Android on large-screen tablets, so a "Notepad" equivalent, ideally linked to the session the notes are taken in, will be useful.
- A way for users to signal chosen friends that they want to eat (at a certain time, in so many minutes, *right now*), including the type of food or restaurant name and seeing if they're also interested.

See Also

The rest of this book shows how to implement most of these functions.

Google Maps has recently started serving building maps. The article shows who to contact to get your building's internal locations added to the map data; if appropriate, consider getting the venue operators to give Google their building's data.

2.8 Using Google Analytics in an Android Application

Ashwini Shahapurkar

Problem

Developers often want to track their applications in terms of features used by users. How can you determine which feature is most used by your app's users?

Solution

Use Google Analytics to track the app based on defined criteria, similar to Google Analytics's website-tracking mechanism.

Discussion

Before we use Google Analytics in our app, we need an analytics account which you can get for free from Google using one of two approaches to getting the Google Analytics SDK running:

Automated Approach

For Android Studio only, you can follow the steps to get the Analytics SDK given at *https://developers.google.com/analytics/devguides/collection/android/resources*, which involve having Google generate a simple configuration file containing your Analytics account, then adding two classpath dependencies and a Gradle plugin in your Gradle build scripts. The plugin will read your downloaded configuration file and apply the information to your code.

Hands-On Approach

A more hands-on approach involves creating your account directly at *https://accounts.google.com/SignUp?continue=https%3A%2F%2Fwww.google.com%2Fanalytics%2Fmobile%2F&hl=en*, then adding two dependencies and providing the analytics account to the SDK. The two dependencies are `com.google.gms:google-services:3.0.0` and `com.google.android.gms:play-services-analytics:10.0.1`.

Now, sign in to your analytics account and create a website profile for the app. The website URL can be fake but should be descriptive. I recommend that you use the reverse package name for this. For example, if the application package name is `com.example.analytics.test`, the website URL for this app can be *http://test.analytics.example.com*. After you create the website profile, a web property ID is generated for that profile. Jot it down - save it in a safe place-as we will be using this ID in our app. The ID, also known as the UA number of the tracking code, uniquely identifies the website profile.

Common Steps

Next, ensure you have the following permissions in your project's *AndroidManifest.xml* file:

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

For both legal and licensing reasons, you *must* inform your users that you are collecting anonymous user data in your app. You can do so via a policy statement, in the end-user license agreement, or somewhere else where users will see this information. See Recipe 2.9.

Now we are ready to track our application. Obtain the singleton instance of the tracker by calling the `GoogleAnalytics.getInstance().newTracker()` method. Usually, you will want to track more than Activities in the app. In such a scenario, it's a good idea to have this tracker instance in the `onCreate()` method of the `Application` class of the app (see Example 2-9).

Example 2-9. The application implementation for tracking

```
public class GADemoApp extends Application {
    /*
     * Define web property ID obtained after creating a profile for the app. If
     * using the Gradle plugin, this should be available as R.xml.global_tracker.
     */
    private String webId = "UA-NNNNNNNN-Y";

    /* Analytics tracker instance */
    Tracker tracker;

    /* This is the getter for the tracker instance. This is called from
     * within the Activity to get a reference to the tracker instance.
     */
    public synchronized Tracker getTracker() {
        if (tracker == null) {
            // Get the singleton Analytics instance, get Tracker from it
            GoogleAnalytics instance = GoogleAnalytics.getInstance(this);
```

```
            // Start tracking the app with your web property ID
            tracker = instance.newTracker(webId);

            // Any app-specific Application setup code goes here...
        }
        return tracker;
    }
}
```

You can track page views and events in the Activity by calling the `setScreenName()` and `send()` methods on the tracker instance (see Example 2-10).

Example 2-10. The Main Activity with tracking

```
public class MainActivity extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // Track the page view for the Activity
        Tracker tracker =
            ((GADemoApp)getApplication()).getTracker();
        tracker.setScreenName("MainActivity");
        tracker.send(new HitBuilders.ScreenViewBuilder().build());

        /* You can track events like button clicks... */
        findViewById(R.id.actionButton).setOnClickListener(v -> {
                Tracker tracker =
                    ((GADemoApp)getApplication()).getTracker();
                tracker.send(new HitBuilders.EventBuilder(
                    "Action Event", "Button Clicked").build());
        });
    }
}
```

Using this mechanism, you can track all the Activities and events inside them. You then visit the Analytics web site to see how many times each Activity or other event has been invoked.

See Also

The main page for the Android Analytics API[*https://developer.android.com/distribute/analyze/start.html*].

Source Download URL

The source code for this example is in the Android Cookbook repository, in the subdirectory *Analytics* (see "Getting and Using the Code Examples" on page 18).

2.9 Setting First-Run Preferences

Ashwini Shahapurkar

Problem

You have an application that collects app usage data anonymously, so you are obligated to make users aware of this the first time they run your application.

Solution

Use shared preferences as persistent storage to store a value, which gets updated only once. Each time the application launches, it will check for this value in the preferences. If the value has been set (is available), it is not the first run of the application; otherwise it is the first run.

Discussion

You can manage the application life cycle by using the `Application` class of the Android framework. We will use shared preferences as persistent storage to store the first-run value.

We will store a Boolean flag in the preferences if this is the first run. When the application is installed and used for the first time, there are no preferences available for it. They will be created for us. In that case the flag will return a value of `true`. After getting the `true` flag, we can update this flag with a value of `false` as we no longer need it to be `true`. See Example 2-11.

Example 2-11. First-run preferences

```
public class MyApp extends Application {

    SharedPreferences mPrefs;

    @Override
    public void onCreate() {
        super.onCreate();

        Context mContext = this.getApplicationContext();
        // 0 = mode private. Only this app can read these preferences.
        mPrefs = mContext.getSharedPreferences("myAppPrefs", 0);


        // Your app initialization code goes here
    }

    public boolean getFirstRun() {
        return mPrefs.getBoolean("firstRun", true);
    }
```

```
    public void setRunned() {
        SharedPreferences.Editor edit = mPrefs.edit();
        edit.putBoolean("firstRun", false);
        edit.commit();
    }
}
```

This flag from the preferences will be tested in the launcher Activity, as shown in Example 2-12.

Example 2-12. Checking whether this is the first run of this app

```
    if(((MyApp) getApplication()).getFirstRun()) {
        // This is the first run
        ((MyApp) getApplication()).setRunned();

        // Your code for the first run goes here

        }
    else {
        // This is not the first run on this device
    }
```

Even if you publish updates for the app and the user installs the updates, these preferences will not be modified; therefore, the code will work for only the first run after installation. Subsequent updates to the app will not bring the code into the picture, unless the user has manually uninstalled and reinstalled the app.

You could use a similar technique for distributing shareware versions of an Android app (i.e., limit the number of trials of the application). In this case, you would use an integer count value in the preferences to indicate the number of trials. Each trial would update the preferences. After the desired value is reached, you would block the usage of the application until the user pays the usage fee.

2.10 Formatting Numbers

Ian Darwin

Problem

You need to format numbers, because the default formatting of `Double.toString()` and friends does not give you enough control over how the results are displayed.

Solution

Use `String.format()` or one of the `NumberFormat` subclasses.

Discussion

The `printf()` function was first included in the C programming language in the 1970s, and it has been used in many other languages since, including Java. Here's a simple `printf()` example in Java SE:

```
System.out.printf("Hello %s at %s%n", userName, time);
```

The preceding example could be expected to print something like this:

```
Hello Robin at Wed Jun 16 08:38:46 EDT 2010
```

Since we don't use `System.out` in Android, you'll be relieved to note that you can get the same string that would be printed, for putting it into a view, by using:

```
String msg = String.format("Hello %s at %s%n", userName, time);
```

If you haven't seen `printf()` before, the first argument is obviously the format code string, and any other arguments here, (`userName` and `time`) are values to be formatted. The format codes begin with a percent sign (`%`) and have at least one "type" code; Table 2-1 shows some common type codes.

Table 2-1. Some common format codes

Character	Meaning
s	String (convert primitive values using defaults; convert objects by `toString`)
d	Decimal integer (`int`, `long`)
f	Floating point (`float`, `double`)
n	Newline
t	Time/date formats, Java-specific; see the discussion referred to in the "See Also" section at the end of the recipe

The default date formatting is pretty ugly, so we often need to expand on it. The `printf()` formatting capabilities are actually housed in the `java.util.Formatter` class, to which reference should be made for the full details of its formatting language.

Unlike `printf()` in other languages you may have used, all these format routines optionally allow you to refer to arguments by their number, by putting a number plus a dollar sign after the `%` lead-in but before the formatting code proper; for example, `%2$3.1f` means to format the second argument as a decimal number with three characters and one digit after the decimal place. This numbering can be used for two purposes: to change the order in which arguments print (often useful with internationalization), and to refer to a given argument more than once. The date/time format char-

acter `t` requires a second character after it, such as `Y` for the year, `m` for the month, and so on. Here we take the `time` argument and extract several fields from it:

```
msg = String.format("Hello at %1$tB %1$td, %1$tY%n", time);
```

This might format as `July 4, 2010`.

To print numbers with a specific precision, you can use `f` with a width and a precision, such as:

```
msg = String.format("Latitude: %10.6f", latitude);
```

This might yield:

```
Latitude: -79.281818
```

While such formatting is OK for specific uses such as latitudes and longitudes, for general use such as currencies, it may give you too much control.

General formatters

Java has an entire package, `java.text`, that is full of formatting routines as general and flexible as anything you might imagine. Like `printf()`, it has an involved formatting language, described in the online documentation page. Consider the presentation of numbers. In North America, the number "one thousand twenty-four and a quarter" is written 1,024.25; in most of Europe it is 1 024,25, and in some other parts of the world it might be written 1.024,25. The formatting of currencies and percentages is equally varied. Trying to keep track of this yourself would drive the average software developer around the bend rather quickly.

Fortunately, the `java.text` package includes a `Locale` class. Furthermore, the Java or Android runtime automatically sets a default `Locale` object based on the user's environment; this code works the same on desktop Java as it does in Android. To provide formatters customized for numbers, currencies, and percentages, the `NumberFormat` class has static factory methods that normally return a `DecimalFormat` with the correct pattern already instantiated. A `DecimalFormat` object appropriate to the user's locale can be obtained from the factory method `NumberFormat.getInstance()` and manipulated using set methods. Surprisingly, the method `setMinimumIntegerDigits()` turns out to be the easy way to generate a number format with leading zeros. Example 2-13 is an example.

Example 2-13. Number formatting demo

```
import java.text.NumberFormat;

/*
 * Format a number our way and the default way.
 */
public class NumFormat2 {
    /** A number to format */
```

```
    public static final double data[] = {
        0, 1, 22d/7, 100.2345678
    };

    public static void main(String[] av) {
        // Get a format instance
        NumberFormat form = NumberFormat.getInstance();

        // Tailor it to look like 999.99[99]
        form.setMinimumIntegerDigits(3);
        form.setMinimumFractionDigits(2);
        form.setMaximumFractionDigits(4);

        // Now print using it
        for (int i=0; i<data.length; i++)
            System.out.println(data[i] + "\tformats as " +
                form.format(data[i]));
    }
}
```

This prints the contents of the array using the `NumberFormat` instance form. We show running it as a main program instead of in an Android application just to isolate the effects of the `NumberFormat`.

For example, `$ java NumFormat2 0.0` formats as `000.00`; with the argument `1.0` it formats as `001.00`, with `3.142857142857143` it formats as `003.1429`, and with `100.2345678` it formats as `100.2346`.

You can also construct a `DecimalFormat` with a particular pattern or change the pattern dynamically using `applyPattern()`. Table 2-2 shows some of the more common pattern characters.

Table 2-2. Common DecimalFormat pattern characters

Character	Explanation
#	Numeric digit (leading zeros suppressed)
0	Numeric digit (leading zeros provided)
.	Locale-specific decimal separator (decimal point)
,	Locale-specific grouping separator (comma in English)
-	Locale-specific negative indicator (minus sign)
%	Shows the value as a percentage
;	Separates two formats: the first for positive and the second for negative values
'	Escapes one of the preceding characters so that it appears as itself
Anything else	Appears as itself

The `NumFormatTest` program uses one `DecimalFormat` to print a number with only two decimal places and a second to format the number according to the default locale, as shown in Example 2-14.

Example 2-14. NumberFormat demo Java SE program

```
import java.text.DecimalFormat;
import java.text.NumberFormat;

public class NumFormatDemo {
    /** A number to format */
    public static final double intlNumber = 1024.25;
    /** Another number to format */
    public static final double ourNumber = 100.2345678;

    public static void main(String[] av) {

        NumberFormat defForm = NumberFormat.getInstance();
        NumberFormat ourForm = new DecimalFormat("##0.##");
        // toPattern() will reveal the combination of #0., etc.
        // that this particular Locale uses to format with
        System.out.println("defForm's pattern is " +
            ((DecimalFormat)defForm).toPattern());
        System.out.println(intlNumber + " formats as " +
            defForm.format(intlNumber));
        System.out.println(ourNumber + " formats as " +
            ourForm.format(ourNumber));
        System.out.println(ourNumber + " formats as " +
            defForm.format(ourNumber) + " using the default format");
    }
}
```

This program prints the given pattern and then formats the same number using several formats:

```
$ java NumFormatTest
defForm's pattern is #,##0.###
1024.25 formats as 1,024.25
100.2345678 formats as 100.23
100.2345678 formats as 100.235 using the default format
```

See Also

Chapter 10 of my book *Java Cookbook* and Part VI of [*Java I/O*] by Elliotte Rusty Harold (both from O'Reilly).

2.11 Formatting with Correct Plurals

Ian Darwin

Problem

You're displaying something like `"Found "+ n +" reviews"`, but in English, "Found 1 reviews" is ungrammatical. You want `"Found 1 review"` for the case `n==1`.

Solution

For simple, English-only results, use a conditional statement. For better results that can be internationalized, use a `ChoiceFormat`. On Android, you can use `<plural>` in an XML resources file.

Discussion

The "quick and dirty" method is to use Java's ternary operator (`cond ? trueval : falseval`) in a string concatenation. Since in English, for most nouns, both zero and plurals get an *s* appended to the noun ("no books, one book, two books"), we need only test for `n==1`:

```
// FormatPlurals.java
public static void main(String argv[]) {
 report(0);
 report(1);
 report(2);
}
/** report -- using conditional operator */
public static void report(int n) {
 System.out.println("Found " + n + " item" + (n==1?"":"s"));
}
```

Running this on Java SE as a main program shows the following output:

```
$ java FormatPlurals
Found 0 items
Found 1 item
Found 2 items
```

The final `println()` statement is short for:

```
if (n==1)
 System.out.println("Found " + n + " item");
else
 System.out.println("Found " + n + " items");
```

This is longer, so Java's ternary conditional operator is worth learning.

Of course, you can't use this arbitrarily, because English is a strange and somewhat idiosyncratic language. Some nouns, such as *bus*, require "es" at the end, while others, such as *cash*, are collective nouns with no plural (you can have two flocks of geese or two stacks of cash, but you cannot have "two geeses" or "two cashes"). Still other nouns, such as *fish*, can be considered plural as they are, although *fishes* is also a correct plural.

A better way

The `ChoiceFormat` class from `java.text` is ideal for handling plurals; it lets you specify singular and plural (or, more generally, range) variations on the noun. It is capable of more, but in Example 2-15 I'll show only a couple of the simpler uses. I specify the

values 0, 1, and 2 (or more), and the string values to print corresponding to each number. The numbers are then formatted according to the range they fall into.

Example 2-15. Formatting plurals using ChoiceFormat

```
import java.text.*;

/**
 * Format a plural correctly, using a ChoiceFormat.
 */
public class FormatPluralsChoice extends FormatPlurals {

    // ChoiceFormat to just give pluralized word
    static double[] limits = { 0, 1, 2 };
    static String[] formats = { "reviews", "review", "reviews"};
    static ChoiceFormat pluralizedFormat =
        new ChoiceFormat(limits, formats);

    // ChoiceFormat to give English text version, quantified
    static ChoiceFormat quantizedFormat = new ChoiceFormat(
        "0#no reviews|1#one review|1<many reviews");

    // Test data
    static int[] data = { -1, 0, 1, 2, 3 };

    public static void main(String[] argv) {
        System.out.println("Pluralized Format");
        for (int i : data) {
            System.out.println("Found " + i + " " +
                pluralizedFormat.format(i));
        }

        System.out.println("Quantized Format");
        for (int i : data) {
            System.out.println("Found " +
                quantizedFormat.format(i));
        }
    }
}
```

Either of these loops generates output similar to the basic version. The code using the `ChoiceFormat` is slightly longer, but more general, and lends itself better to internationalization. Put the string for the "quantized" form constructor into *strings.xml* and it will be part of your localization actions.

The best way (Android only)

Create a file in */res/values$$/* containing something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <plurals name="numberOfSongsAvailable">
  <item quantity="one">One item found.</item>
```

```
    <item quantity="other">%d items found.</item>
    </plurals>
</resources>
```

In your code, you can then use the following:

```
int count = getNumberOfsongsAvailable();
Resources res = getResources();
String songsFound =
    res.getQuantityString(R.plurals.numberOfSongsAvailable, count);
```

This use of XML resources was suggested by Tomas Persson.

See Also

For the Android-specific way, see the developer documentation on quantity strings.

Source Download URL

You can download the source code for this example from GitHub.

2.12 Formatting the Time and Date for Display

Pratik Rupwal

Problem

You want to display the time and date in different standard formats.

Solution

The `DateFormat` class provides APIs for formatting time and date in a custom format. Using these APIs requires minimal effort.

Discussion

Example 2-16 adds five different `TextViews` for showing the time and date in different formats.

Example 2-16. The TextView layout

```
<?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"
    >
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
```

```
    android:id="@+id/textview1"
    />
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:id="@+id/textview2"
    />
    <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:id="@+id/textview3"
    />
    <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:id="@+id/textview4"
    />
    <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:id="@+id/textview5"
    />

</LinearLayout>
```

Example 2-17 obtains the current time and date using the `java.util.Date` class and then displays it in different formats (please refer to the comments for sample output).

Example 2-17. The date formatter Activity

```
package com.sym.dateformatdemo;

import java.util.Date;
import android.app.Activity;
import android.os.Bundle;
import android.text.format.DateFormat;
import android.widget.TextView;

public class TestDateFormatterActivity extends Activity {
    /** Called when the Activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        TextView textView1 = (TextView) findViewById(R.id.textview1);
        TextView textView2 = (TextView) findViewById(R.id.textview2);
        TextView textView3 = (TextView) findViewById(R.id.textview3);
        TextView textView4 = (TextView) findViewById(R.id.textview4);
        TextView textView5 = (TextView) findViewById(R.id.textview5);

        String delegate = "MM/dd/yy hh:mm a"; // 09/21/2011 02:17 pm
        Date noteTS = new Date();
        textView1.setText("Found Time :: "+DateFormat.format(delegate,noteTS));
```

```
        delegate = "MMM dd, yyyy h:mm aa"; // Sep 21,2011 02:17 pm
        textView2.setText("Found Time :: "+DateFormat.format(delegate,noteTS));

        delegate = "MMMM dd, yyyy h:mmaa"; // September 21,2011 02:17pm
        textView3.setText("Found Time :: "+DateFormat.format(delegate,noteTS));

        delegate = "E, MMMM dd, yyyy h:mm:ss aa";//Wed, September 21,2011 02:17:48 pm
        textView4.setText("Found Time :: "+DateFormat.format(delegate,noteTS));

        delegate =
            "EEEE, MMMM dd, yyyy h:mm aa"; //Wednesday, September 21,2011 02:17:48 pm
        textView5.setText("Found Time :: "+DateFormat.format(delegate,noteTS));
    }
}
```

See Also

Recipe 2.13. Also, the classes shown in the following table, in the package `android.text.format`, may be of use in this type of application.

Name	Usage
`DateUtils`	This class contains various date-related utilities for creating text for things like elapsed time and date ranges, strings for days of the week and months, and a.m./p.m. text.
`Formatter`	This is a utility class to aid in formatting common values that are not covered by `java.util.Formatter`.
`Time`	This class is a faster replacement for the `java.util.Calendar` and `java.util.GregorianCalendar` classes.

Source Download URL

The source code for this example is in the Android Cookbook repository, in the subdirectory *DateFormatDemo* (see "Getting and Using the Code Examples" on page 18).

2.13 Simplifying Date/Time Calculations with the Java 8 java.time API

Ian Darwin

Problem

You've heard that the JSR-310 date/time API, included in Java SE 8, simplifies date and time calculations, and you'd like to use it in Android.

Solution

You can use the new java.time API in Android O and later. Since Android did not become fully compliant with JDK 8 even in Android Nougat, despite being "based on" OpenJDK 8, for Android Nougat and earlier, you must use a third-party library

such as the JSR-310 "backport" to access the `java.time` facilities, albeit with a different package name.

Discussion

There is a long history to the `java.time` API that I won't bore you with here; suffice it to say that we are all indebted to Steven Colbourne for inventing it and for his constancy in urging first Sun, then Oracle, to incorporate it into Java, which finally happened in Java 8. For licensing reasons, the backport of JSR-310—by its original author—to Java 6/7 was placed in a non-Java package, `org.threeten.bp`.

Since Android N didn't provide full compatibility with Java 8, we use an external library. We'll use an Android-specific version of this "backport" library, by Jake Wharton, is available on GitHub. You can add it to any Gradle or Maven project just by adding the coordinates `compile 'com.jakewharton.threetenabp:threetenabp:1.0.3'` to your build script (the version number may change over time, of course).

Here is an example to show you the level of complexity of the kinds of calculations that are built in. I've omitted the imports because they differ from the backport libraries and "standard Java" and Android O. The example shows how little code is needed to figure out the day of the month on which the next weekly and monthly paydays occur:

```
LocalDateTime now = LocalDateTime.now();
LocalDateTime weeklyPayDay =
    now.with(TemporalAdjusters.next(DayOfWeek.FRIDAY));
weekly.setText("Weekly employees' payday is Friday " +
    weeklyPayDay.getMonth() + " " +
    weeklyPayDay.getDayOfMonth());
LocalDateTime monthlyPayDay =
    now.with(TemporalAdjusters.lastInMonth(DayOfWeek.FRIDAY));
monthly.setText("Monthly employees are paid on " +
    monthlyPayDay.getMonth() + " " +
    monthlyPayDay.getDayOfMonth());
```

The API includes `LocalDate` objects, which just represent one particular day; `LocalTime` objects, which represent a time of day; and `LocalDateTime` objects, which represent a date and a time. As the names imply, all three are local, not meant to represent time across the world's time zones. For that, you want to use one of several classes that represent time zones. See the `java.time` documentation for details of all the classes.

To use the backport library on Android N and earlier, you need one extra call to initialize it, either in your `Application` class (see Recipe 2.3) or in your Activity. In the main Activity's `onCreate()` method you'd say:

```
AndroidThreeTen.init(getApplication());
```

The result should look like Figure 2-4.

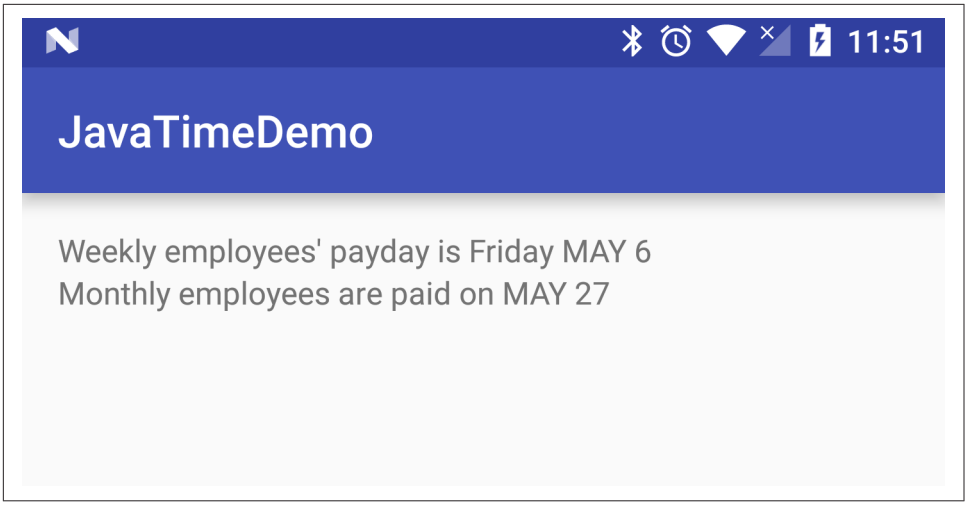


Figure 2-4. Java time example

See Also

The new API is covered in Chapter 6 of my *Java Cookbook* and in some tutorials on the web. Make sure you use the version of the tutorial corresponding to the API you are using. The Java 8 version differs slightly from "ThreeTen" versions, and these both differ from the original Joda Time versions.

Source Download URL

The source code for this example is in the Android Cookbook repository, in the subdirectory *JavaTimeDemo* (see "Getting and Using the Code Examples" on page 18).

2.14 Controlling Input with KeyListeners

Pratik Rupwal

Problem

Your application contains text boxes in which you want to restrict users to entering only numbers; also, in some cases you want to allow only positive numbers, or integers, or dates.

Solution

Android provides `KeyListener` classes to help you restrict users to entering only numbers, positive numbers, integers, positive integers, and much more.

Discussion

The `Android.text.method` package includes a `KeyListener` interface, along with some classes such as `DigitsKeyListener` and `DateKeyListener` that implement this interface.

Example 2-18 is a sample application that demonstrates a few of these classes. This layout file creates five `TextViews` and five `EditTexts`; the `TextViews` display the input type allowed for their respective `EditTexts`.

Example 2-18. Layout with TextViews and EditTexts

```
<?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"
    >

    <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:id="@+id/textview1"
    android:text="digits listener with signs and decimal points"
    />
    <EditText
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:id="@+id/editText1"
    />

    <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:id="@+id/textview2"
    android:text="digits listener without signs and decimal points"
    />
    <EditText
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:id="@+id/editText2"
    />

    <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:id="@+id/textview3"
    android:text="date listener"
    />
    <EditText
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:id="@+id/editText3"
    />
```

```
    <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:id="@+id/textview4"
    android:text="multitap listener"
    />
    <EditText
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:id="@+id/editText4"
    />

    <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:id="@+id/textview5"
    android:text="qwerty listener"
    />
    <EditText
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:id="@+id/editText5"
    />
</LinearLayout>
```

Example 2-19 is the code for the Activity that restricts the `EditText` inputs to numbers, positive integers, and so on (refer to the comments for groups of keys allowed).

Example 2-19. The main Activity

```
import android.app.Activity;
import android.os.Bundle;
import android.text.method.DateKeyListener;
import android.text.method.DigitsKeyListener;
import android.text.method.MultiTapKeyListener;
import android.text.method.QwertyKeyListener;
import android.text.method.TextKeyListener;
import android.widget.EditText;

public class KeyListenerDemo extends Activity {
    /** Called when the Activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        // Allows digits with positive/negative signs and decimal points
        EditText editText1=(EditText)findViewById(R.id.editText1);
        DigitsKeyListener digkl1=DigitsKeyListener.getInstance(true,true);
        editText1.setKeyListener(digkl1);

        // Allows positive integers only (no decimal values allowed)
        EditText editText2=(EditText)findViewById(R.id.editText2);
        DigitsKeyListener digkl2=DigitsKeyListener.getInstance();
        editText2.setKeyListener(digkl2);
```

```
        // Allows dates only
        EditText editText3=(EditText)findViewById(R.id.editText3);
        DateKeyListener dtkl=new DateKeyListener();
        editText3.setKeyListener(dtkl);

        // Allows multitap with 12-key keypad layout
        EditText editText4=(EditText)findViewById(R.id.editText4);
        MultiTapKeyListener multitapkl =
            new MultiTapKeyListener(TextKeyListener.Capitalize.WORDS,true);
        editText4.setKeyListener(multitapkl);

        // Allows qwerty layout for typing
        EditText editText5=(EditText)findViewById(R.id.editText5);
        QwertyKeyListener qkl =
            new QwertyKeyListener(TextKeyListener.Capitalize.SENTENCES,true);
        editText5.setKeyListener(qkl);
    }
}
```

To use `MultiTapKeyListener`, your phone should support the 12-key layout and it needs to be activated. To activate the 12-key layout, go to Settings → Language and Keyboard → On-screen Keyboard Layout and then select the "Phone layout" options.

See Also

The `Listener` types in the following table will be of use in writing this type of application.

Name	Usage
`BaseKeyListener`	This is an abstract base class for key listeners.
`DateTimeKeyListener`	This is for entering dates and times in the same text field.
`MetaKeyKeyListener`	This base class encapsulates the behavior for tracking the state of meta keys such as SHIFT, ALT, and SYM, as well as the pseudo-meta state of selecting text.
`NumberKeyListener`	This is for numeric text entry.
`TextKeyListener`	This is the key listener for typing normal text.
`TimeKeyListener`	This is for entering times in a text field.

2.15 Backing Up Android Application Data

Pratik Rupwal

Problem

When a user performs a factory reset or converts to a new Android-powered device, the application loses stored data or application settings.

Solution

Android's Backup Manager helps to automatically restore backup data or application settings when the application is reinstalled.

Discussion

Android's Backup Manager basically operates in two modes: backup and restore. During a backup operation, the Backup Manager (`BackupManager` class) queries your application for backup data, then hands it to a backup transport, which then delivers the data to cloud-based storage. During a restore operation, the Backup Manager retrieves the backup data from the backup transport and returns it to your application so that your application can restore the data to the device. It's possible for your application to request a restore, but not necessary because Android performs a restore operation when your application is installed and backup data associated with the user exists. The primary scenario in which backup data is restored happens when a user resets her device or upgrades to a new device and her previously installed applications are reinstalled.

Example 2-20 shows how to implement the Backup Manager for your application so that you can save the current state of your application.

Example 2-20. The backup/restore layout

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

    <ScrollView
       android:orientation="vertical"
       android:layout_width="fill_parent"
       android:layout_height="fill_parent"
       android:layout_weight="1">

        <LinearLayout
           android:orientation="vertical"
           android:layout_width="match_parent"
           android:layout_height="wrap_content">

            <TextView android:text="@string/filling_text"
               android:textSize="20dp"
               android:layout_marginTop="20dp"
               android:layout_marginBottom="10dp"
               android:layout_width="match_parent"
               android:layout_height="wrap_content"/>

            <RadioGroup android:id="@+id/filling_group"
               android:layout_width="match_parent"
               android:layout_height="wrap_content"
```

```
            android:layout_marginLeft="20dp"
            android:orientation="vertical">

             <RadioButton android:id="@+id/bacon"
                android:text="@string/bacon_label"/>
             <RadioButton android:id="@+id/pastrami"
                android:text="@string/pastrami_label"/>
             <RadioButton android:id="@+id/hummus"
                android:text="@string/hummus_label"/>

         </RadioGroup>

         <TextView android:text="@string/extras_text"
            android:textSize="20dp"
            android:layout_marginTop="20dp"
            android:layout_marginBottom="10dp"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"/>

         <CheckBox android:id="@+id/mayo"
            android:text="@string/mayo_text"
            android:layout_marginLeft="20dp"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"/>

         <CheckBox android:id="@+id/tomato"
            android:text="@string/tomato_text"
            android:layout_marginLeft="20dp"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"/>

      </LinearLayout>

   </ScrollView>

</LinearLayout>
```

Here is a basic description of the procedure in step-by-step form:

1. Create a `BackupManagerExample` project in Eclipse.
2. Open the *layout/backup_restore.xml* file and insert the code in Example 2-20 into it.
3. Open the *values/string.xml* file and insert into it the code shown in Example 2-21.
4. Your manifest file will look like the code shown in Example 2-22.
5. The code in Example 2-23 completes the implementation of the Backup Manager for your application.

Example 2-21. Strings for the example

```
<resources>
  <string name="hello">Hello World, BackupManager!</string>
  <string name="app_name">BackupManager</string>
  <string name="filling_text">Choose Settings for your application:</string>
  <string name="bacon_label">Sound On</string>
  <string name="pastrami_label">Vibration On</string>
  <string name="hummus_label">Backlight On</string>
  <string name="extras_text">Extras:</string>
  <string name="mayo_text">Use Orientation?</string>
  <string name="tomato_text">Use Camera?</string>
</resources>
```

Example 2-22. AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.sym.backupmanager"
      android:versionCode="1"
      android:versionName="1.0">
    <uses-sdk android:minSdkVersion="9" />

    <application android:label="Backup/Restore" android:icon="@drawable/icon"
        android:backupAgent="ExampleAgent"> <!--Here you specify the backup agent-->

        <!--Some backup transports may require API keys or other metadata-->
        <meta-data android:name="com.google.android.backup.api_key"
                android:value="INSERT YOUR API KEY HERE" />

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

</manifest>
```

Example 2-23. The backup/restore Activity

```
package com.sym.backupmanager;

import android.app.Activity;
import android.app.backup.BackupManager;
import android.os.Bundle;
import android.util.Log;
import android.widget.CheckBox;
import android.widget.CompoundButton;
import android.widget.RadioGroup;
import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
```

```
public class BackupManagerExample extends Activity {
    static final String TAG = "BRActivity";

    static final Object[] sDataLock = new Object[0];

    static final String DATA_FILE_NAME = "saved_data";

    RadioGroup mFillingGroup;
    CheckBox mAddMayoCheckbox;
    CheckBox mAddTomatoCheckbox;

    File mDataFile;

    BackupManager mBackupManager;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.backup_restore);

        mFillingGroup = (RadioGroup) findViewById(R.id.filling_group);
        mAddMayoCheckbox = (CheckBox) findViewById(R.id.mayo);
        mAddTomatoCheckbox = (CheckBox) findViewById(R.id.tomato);

         mDataFile = new File(getFilesDir(), BackupManagerExample.DATA_FILE_NAME);

         mBackupManager = new BackupManager(this);

        populateUI();
    }

    void populateUI() {
        RandomAccessFile file;

        int whichFilling = R.id.pastrami;
        boolean addMayo = false;
        boolean addTomato = false;

        synchronized (BackupManagerExample.sDataLock) {
            boolean exists = mDataFile.exists();
            try {
                file = new RandomAccessFile(mDataFile, "rw");
                if (exists) {
                    Log.v(TAG, "datafile exists");
                    whichFilling = file.readInt();
                    addMayo = file.readBoolean();
                    addTomato = file.readBoolean();
                    Log.v(TAG, "  mayo=" + addMayo
                            + " tomato=" + addTomato
                            + " filling=" + whichFilling);
                } else {
                    Log.v(TAG, "creating default datafile");
                    writeDataToFileLocked(file,
                            addMayo, addTomato, whichFilling);
```

```
                mBackupManager.dataChanged();
            }
        } catch (IOException ioe) {
            // Do some error handling here!
        }
    }

    mFillingGroup.check(whichFilling);
    mAddMayoCheckbox.setChecked(addMayo);
    mAddTomatoCheckbox.setChecked(addTomato);

    mFillingGroup.setOnCheckedChangeListener(
            new RadioGroup.OnCheckedChangeListener() {
                public void onCheckedChanged(RadioGroup group,
                        int checkedId) {
                    Log.v(TAG, "New radio item selected: " + checkedId);
                    recordNewUIState();
                }
            });

    CompoundButton.OnCheckedChangeListener checkListener
            = new CompoundButton.OnCheckedChangeListener() {
        public void onCheckedChanged(CompoundButton buttonView,
                boolean isChecked) {
            Log.v(TAG, "Checkbox toggled: " + buttonView);
            recordNewUIState();
        }
    };
    mAddMayoCheckbox.setOnCheckedChangeListener(checkListener);
    mAddTomatoCheckbox.setOnCheckedChangeListener(checkListener);
}

void writeDataToFileLocked(RandomAccessFile file,
        boolean addMayo, boolean addTomato, int whichFilling)
    throws IOException {
        file.setLength(0L);
        file.writeInt(whichFilling);
        file.writeBoolean(addMayo);
        file.writeBoolean(addTomato);
        Log.v(TAG, "NEW STATE: mayo=" + addMayo
                + " tomato=" + addTomato
                + " filling=" + whichFilling);
}

void recordNewUIState() {
    boolean addMayo = mAddMayoCheckbox.isChecked();
    boolean addTomato = mAddTomatoCheckbox.isChecked();
    int whichFilling = mFillingGroup.getCheckedRadioButtonId();
    try {
        synchronized (BackupManagerExample.sDataLock) {
            RandomAccessFile file = new RandomAccessFile(mDataFile, "rw");
            writeDataToFileLocked(file, addMayo, addTomato, whichFilling);
        }
    } catch (IOException e) {
        Log.e(TAG, "Unable to record new UI state");
    }
```

```
        mBackupManager.dataChanged();
    }
}
```

Data backup is not guaranteed to be available on all Android-powered devices. However, your application is not adversely affected in the event that a device does not provide a backup transport. If you believe that users will benefit from data backup in your application, you can implement it as described in this recipe, test it, and then publish your application without any concern about which devices actually perform backups. When your application runs on a device that does not provide a backup transport, the application will operate normally but will not receive callbacks from the Backup Manager to back up data.

Although you cannot know what the current transport is, you are always assured that your backup data cannot be read by other applications on the device. Only the Backup Manager and backup transport have access to the data you provide during a backup operation.

Because the cloud storage and transport services can differ among devices, Android makes no guarantees about the security of your data while using backup. You should always be cautious about using backup to store sensitive data, such as usernames and passwords.

Testing your backup agent

Once you've implemented your backup agent, you can use the `bmgr` command to test the backup and restore functionality by following these steps:

1. Install your application on a suitable Android system image, running any current emulator or device with Google Play Services.
2. Ensure that backup capability is enabled. If you are using the emulator, you can enable backup with the following command from your SDK tools/path:

   ```
   $ adb shell bmgr enable true
   ```

3. If you are using a device, open the system settings, select Privacy, and then enable "Back up my data" and "Automatic restore."
4. Open your application and initialize some data.

   If you've properly implemented backup capability in your application, it should request a backup each time the data changes. For example, each time the user changes some data, your app should call `dataChanged()`, which adds a backup request to the Backup Manager queue. For testing purposes, you can also make a request with the following ++bmgr++ command:

```
$ adb shell bmgr backup your.package.name
```

5. Initiate a backup operation:

   ```
   $ adb shell bmgr run
   ```

   This forces the Backup Manager to perform all backup requests that are in its queue.

6. Uninstall your application:

   ```
   $ adb uninstall your.package.name
   ```

7. Reinstall your application.

   If your backup agent is successful, all the data you initialized in step 4 is restored.

2.16 Using Hints Instead of Tool Tips

Daniel Fowler

Problem

Android devices can have small screens, so there may not be room for help text, and tool tips are not part of the platform.

Solution

Android provides the `hint` attribute for `Views`.

Discussion

Sometimes an input field needs clarification with regard to the value that should be entered. For example, a stock-ordering application asking for item quantities may need to state the minimum order size. In desktop programs, with large screens and the use of a mouse, extra messages can be displayed in the form of tool tips (a pop-up label over a field when the mouse moves over it). Alternatively, long descriptive labels may be used. With Android devices, the screen may be small and no mouse is generally used. The alternative here is to use the `android:hint` attribute on a `View`. This causes a "watermark" containing the hint text to be displayed in the input field when it is empty; this disappears when the user starts typing in the field. The corresponding function for `android:hint` is `setHint(int resourceId)`. Figure 2-5 shows an example hint.

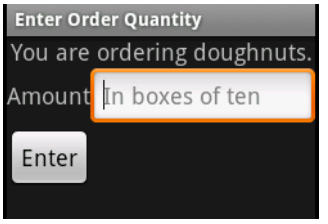


Figure 2-5. An example with hints

You can set the color of the hint text with `android:textColorHint`, with `setHintTextColor(int color)` being the associated function.

Using hints can also help with screen layouts when space is tight. A screen design can sometimes be improved by removing a label and using a hint, as shown in Figure 2-6.

The `EditText` definition in Figure 2-6 is shown in the following code so that you can see `android:hint` in use:

```
<EditText android:id="@+id/etQuantity"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:hint="Number of boxes of ten"
    android:textSize="18sp"/>
```

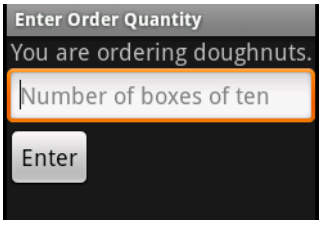


Figure 2-6. A hint and no label

Hints can guide users as they are filling in app fields, though as with any feature overuse is possible. Hints should not be used when it is obvious what is required; a field with a label of "First Name" would not need a hint such as "Enter your first name here," for example. Figure 2-6 shows our hypothetical ordering application improved somewhat by removing the redundant label.