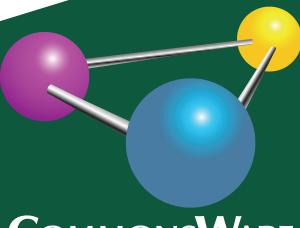


Version
3.1
*for Android
2.3*

Android™ Programming Tutorials

Mark L. Murphy



COMMONSWARE

Android Programming Tutorials

by Mark L. Murphy

Android Programming Tutorials
by Mark L. Murphy

Copyright © 2009-2011 CommonsWare, LLC. All Rights Reserved.
Printed in the United States of America.

CommonsWare books may be purchased in printed (bulk) or digital form for educational or business use. For more information, contact direct@commonsware.com.

Printing History:

Jan 2011: Version 3.1 ISBN: 978-0-9816780-4-7

The CommonsWare name and logo, “Busy Coder’s Guide”, and related trade dress are trademarks of CommonsWare, LLC.

All other trademarks referenced in this book are trademarks of their respective firms.

The publisher and author(s) assume no responsibility for errors or omissions or for damages resulting from the use of the information contained herein.

Table of Contents

Welcome to the Warescription!	xvii
Preface	xix
Welcome to the Book!	xix
Prerequisites	xix
Using the Tutorials	xx
Warescription	xxii
What's New	xxii
About the "Further Reading" Sections	xxiii
Errata and Book Bug Bounty	xxiii
Source Code License	xxiv
Creative Commons and the Four-to-Free (42F) Guarantee	xxiv
Lifecycle of a CommonsWare Book	xxv
Roster of Tutorials	xxvi
Your First Android Project	1
Step #1: Create the New Project	1
Step #1: Eclipse	2
Step #2: Command Line	5
Step #2: Build, Install, and Run the Application in Your Emulator or Device	6

Step #1: Eclipse.....	6
Step #2: Command Line.....	7
A Simple Form.....	11
Step-By-Step Instructions.....	11
Step #1: Generate the Application Skeleton.....	11
Step #2: Modify the Layout.....	12
Step #3: Compile and Install the Application.....	13
Step #4: Run the Application in the Emulator.....	14
Step #5: Create a Model Class.....	14
Step #6: Save the Form to the Model.....	15
Extra Credit.....	16
Further Reading.....	17
A Fancier Form.....	19
Step-By-Step Instructions.....	19
Step #1: Switch to a TableLayout.....	19
Step #2: Add a RadioGroup.....	21
Step #3: Update the Model.....	23
Step #4: Save the Type to the Model.....	24
Extra Credit.....	26
Further Reading.....	27
Adding a List.....	29
Step-By-Step Instructions.....	29
Step #1: Hold a List of Restaurants.....	29
Step #2: Save Adds to List.....	30
Step #3: Implement toString().....	31
Step #4: Add a ListView Widget.....	31
Step #5: Build and Attach the Adapter.....	33

Extra Credit.....	35
Further Reading.....	36
Making Our List Be Fancy.....	37
Step-By-Step Instructions.....	37
Step #1: Create a Stub Custom Adapter.....	38
Step #2: Design Our Row.....	38
Step #3: Override getView(): The Simple Way.....	40
Step #4: Create a RestaurantHolder.....	41
Step #5: Recycle Rows via RestaurantHolder.....	42
Extra Credit.....	45
Further Reading.....	45
Splitting the Tab.....	47
Step-By-Step Instructions.....	47
Step #1: Rework the Layout.....	47
Step #2: Wire In the Tabs.....	49
Step #3: Get Control On List Events.....	51
Step #4: Update Our Restaurant Form On Clicks.....	51
Step #5: Switch Tabs On Clicks.....	52
Extra Credit.....	56
Further Reading.....	56
Menus and Messages.....	57
Step-By-Step Instructions.....	57
Step #1: Add Notes to the Restaurant.....	57
Step #2: Add Notes to the Detail Form.....	58
Step #3: Define the Option Menu.....	60
Step #4: Show the Notes as a Toast.....	61
Extra Credit.....	67

Further Reading.....	67
Sitting in the Background.....	69
Step-By-Step Instructions.....	69
Step #1: Initialize the Progress Bar.....	69
Step #2: Create the Work Method.....	70
Step #3: Fork the Thread from the Menu.....	71
Step #4: Manage the Progress Bar.....	73
Extra Credit.....	75
Further Reading.....	76
Life and Times.....	77
Step-By-Step Instructions.....	77
Step #1: Lengthen the Background Work.....	77
Step #2: Pause in onPause().....	78
Step #3: Resume in onResume().....	79
Extra Credit.....	86
Further Reading.....	86
A Few Good Resources.....	87
Step-By-Step Instructions.....	87
Step #1: Review our Current Resources.....	87
Step #2: Create a Landscape Layout.....	88
Extra Credit.....	91
Further Reading.....	92
The Restaurant Store.....	93
Step-By-Step Instructions.....	93
Step #1: Create a Stub SQLiteOpenHelper.....	93
Step #2: Manage our Schema.....	94
Step #3: Remove Extraneous Code from LunchList.....	95

Step #4: Get Access to the Helper.....	95
Step #5: Save a Restaurant to the Database.....	96
Step #6: Get the List of Restaurants from the Database.....	98
Step #7: Change our Adapter and Wrapper.....	99
Step #8: Clean Up Lingering ArrayList References.....	101
Step #9: Refresh Our List.....	102
Extra Credit.....	107
Further Reading.....	107
Getting More Active.....	109
Step-By-Step Instructions.....	109
Step #1: Create a Stub Activity.....	109
Step #2: Launch the Stub Activity on List Click.....	110
Step #3: Move the Detail Form UI.....	111
Step #4: Clean Up the Original UI.....	114
Step #5: Pass the Restaurant _ID.....	115
Step #6: Load the Restaurant Into the Form.....	116
Step #7: Add an "Add" Menu Option.....	118
Step #8: Detail Form Supports Add and Edit.....	119
Extra Credit.....	133
Further Reading.....	133
What's Your Preference?.....	135
Step-By-Step Instructions.....	135
Step #1: Define the Preference XML.....	135
Step #2: Create the Preference Activity.....	136
Step #3: Connect the Preference Activity to the Option Menu.....	137
Step #4: Apply the Sort Order on Startup.....	141
Step #5: Listen for Preference Changes.....	142

Step #6: Re-Apply the Sort Order on Changes.....	143
Extra Credit.....	144
Further Reading.....	145
Turn, Turn, Turn.....	147
Step-By-Step Instructions.....	147
Step #1: Add a Stub onSaveInstanceState().....	147
Step #2: Pour the Form Into the Bundle.....	148
Step #3: Repopulate the Form.....	148
Step #4: Fix Up the Landscape Detail Form.....	149
Extra Credit.....	150
Further Reading.....	151
Raising (Something Like) a Tweet.....	155
Step-By-Step Instructions.....	155
Step #1: Set Up an Identi.ca Account.....	155
Step #2: Create a Stub Application and Activity.....	156
Step #3: Create a Layout.....	156
Step #4: Listen for Send Actions.....	158
Step #5: Make the Status Post Request.....	159
Extra Credit.....	162
Further Reading.....	163
Opening a JAR.....	165
Step-By-Step Instructions.....	165
Step #1: Obtain the JTtwitter JAR.....	165
Step #2: Switch from HttpClient to JTtwitter.....	166
Step #3: Create Preferences for Account Information.....	167
Step #4: Use Account Information from Preferences.....	170
Extra Credit.....	174

Further Reading.....	.175
Listening To Your Friends.....	.177
Step-By-Step Instructions.....	.177
Step #1: Create a Service Stub.....	.177
Step #2: Set Up a Background Thread.....	.178
Step #3: Poll Your Friends.....	.179
Step #4: Find New Statuses.....	.180
Step #5: Set up the Public API.....	.181
Extra Credit.....	.182
Further Reading.....	.182
No, Really Listening To Your Friends.....	.183
Step-By-Step Instructions.....	.183
Step #1: Define the Callback.....	.183
Step #2: Enable Callbacks in the Service.....	.184
Step #3: Manage the Service and Register the Account.....	.186
Step #4: Display the Timeline.....	.188
Extra Credit.....	.200
Further Reading.....	.201
Your Friends Seem Remote.....	.203
Step-By-Step Instructions.....	.203
Step #1: Back Up or Branch Your Project.....	.203
Step #2: Create a Fresh Project.....	.204
Step #3: Move the Service to the New Project.....	.204
Step #4: Implement and Copy the AIDL.....	.205
Step #5: Implement the Client Side.....	.206
Step #6: Implement the Service Side.....	.208
Step #7: Restore Your Project.....	.217

Extra Credit.....	217
Further Reading.....	217
A Subtle Notification.....	219
Step-By-Step Instructions.....	219
Step #1: Pick a Word and Icon.....	219
Step #2: Raise the Notification.....	220
Step #3: Watch for the Keyword.....	221
Step #4: Clearing the Notification.....	223
Extra Credit.....	225
Further Reading.....	225
Posts On Location.....	227
Step-By-Step Instructions.....	227
Step #1: Get the LocationManager.....	227
Step #2: Register for Location Updates.....	228
Step #3: Add "Insert Location" Menu.....	229
Step #4: Insert the Last Known Location.....	230
Extra Credit.....	232
Further Reading.....	233
Here a Post, There a Post.....	235
Step-By-Step Instructions.....	235
Step #1: Register for a Map API Key.....	235
Step #2: Create a Basic MapActivity.....	236
Step #3: Launch the Map on Location-Bearing Status Click.....	237
Step #4: Show the Location Via a Pin.....	240
Extra Credit.....	243
Further Reading.....	243

Media.....	245
Step-By-Step Instructions.....	245
Step #1: Obtain and Install a Video Clip.....	245
Step #2: Create the Stub Helpcast Activity.....	246
Step #3: Launch the Helpcast from the Menu.....	247
Extra Credit.....	249
Further Reading.....	250
Browsing Some Posts.....	251
Step-By-Step Instructions.....	251
Step #1: Add Auto-Linking.....	251
Step #2: Draft and Package the Help HTML.....	252
Step #3: Create a Help Activity.....	252
Step #4: Splice In the Help Activity.....	253
Extra Credit.....	255
Further Reading.....	256
High-Priced Help.....	257
Step-By-Step Instructions.....	257
Step #1: Enable Javascript.....	257
Step #2: Create the Java Object to Inject.....	258
Step #3: Inject the Java Object.....	258
Step #4: Leverage the Java Object from Javascript.....	259
Extra Credit.....	260
Further Reading.....	260
Now Your Friends Seem Animated.....	263
Step-By-Step Instructions.....	263
Step #1: Set Up the Option Menu.....	263
Step #2: Show and Hide the Status Entry Widgets.....	266

Step #3: Fading In and Out.....	268
Extra Credit.....	270
Further Reading.....	270
Messages From The Great Beyond.....	271
Step-By-Step Instructions.....	271
Step #1: Broadcast the Intent.....	271
Step #2: Catch and Use the Intent.....	273
Extra Credit.....	274
Further Reading.....	274
Contacting Our Friends.....	275
Step-By-Step Instructions.....	275
Step #1: Fake the Contact Data.....	275
Step #2: Design the Highlight.....	280
Step #3: Find and Highlight Matching Contacts.....	281
Extra Credit.....	282
Further Reading.....	283
Android Would Like Your Attention.....	285
Step-By-Step Instructions.....	285
Step #1: Track the Battery State.....	285
Step #2: Use the Battery State.....	287
Extra Credit.....	288
Further Reading.....	288
Now, Your Friends Are Alarmed.....	289
Step-By-Step Instructions.....	289
Step #1: Import a Reusable Component.....	289
Step #2: Create the Alarm BroadcastReceiver.....	290
Step #3: Doing the Work.....	291

Extra Credit.....	298
Further Reading.....	298
Searching For Food.....	299
Step-By-Step Instructions.....	299
Step #1: Have the List Conduct the Search.....	299
Step #2: Integrate the Search in the Application.....	302
Extra Credit.....	304
Further Reading.....	304
Look Inside Yourself.....	305
Step-By-Step Instructions.....	305
Step #1: Create a Stub Project.....	305
Step #2: Create a Layout.....	306
Step #3: Find the Correct Contact Uri.....	307
Step #4: Attach the Button to the Contacts.....	307
Step #5: Populate the List.....	308
Extra Credit.....	312
Further Reading.....	313
A Restaurant In Your Own Home.....	315
Step-By-Step Instructions.....	315
Step #1: Find An App Widget Background and Icon.....	315
Step #2: Design the App Widget Layout.....	316
Step #3: Add an (Empty) AppWidgetProvider.....	316
Step #4: Add the Widget Metadata.....	317
Step #5: Update the Manifest.....	318
Step #6: Show a Random Restaurant.....	321
Extra Credit.....	324
Further Reading.....	324

More Home Cooking.....	325
Step-By-Step Instructions.....	325
Step #1: Find a Button Graphic.....	325
Step #2: Add the Button to the Layout.....	326
Step #3: Migrate Update Logic to an IntentService.....	327
Step #4: Get Control on Button Clicks.....	330
Step #5: Get Control on Name Clicks.....	331
Extra Credit.....	333
Further Reading.....	333
Take a Monkey to Lunch.....	335
Step-By-Step Instructions.....	335
Step #1: Prep LunchList.....	335
Step #2: Run the Monkey.....	335
Extra Credit.....	336
Further Reading.....	337
Asking Permission to Place a Call.....	339
Step-By-Step Instructions.....	339
Step #1: Add a Phone Number to the Database Schema.....	339
Step #2: Intelligently Handle Database Updates.....	340
Step #3: Add Phone Number Support to the Rest of the Helper.....	340
Step #4: Collect the Phone Number on the Detail Form.....	342
Step #5: Ask for Permission to Make Calls.....	343
Step #6: Dial the Number.....	344
Step #7: Make the Call.....	346
Extra Credit.....	347
Further Reading.....	347

Photographic Memory.....	349
Step-By-Step Instructions.....	349
Step #1: Adjust the Manifest.....	349
Step #2: Create the Photographer Layout.....	351
Step #3: Create the Photographer Class.....	352
Step #4: Tie In the Photographer Class.....	354
Extra Credit.....	355
Further Reading.....	355
Sensing a Disturbance.....	357
Step-By-Step Instructions.....	357
Step #1: Implement a Shaker.....	357
Step #2: Hook Into the Shaker.....	360
Step #3: Make a Random Selection on a Shake.....	361
Extra Credit.....	362
Further Reading.....	362
Getting the Word Out.....	363
Step-By-Step Instructions.....	363
Step #1: Add a "Send SMS" Option Menu.....	364
Step #2: Find Contacts' Mobile Numbers.....	365
Step #3: Pick a Person.....	366
Step #4: Send the Message.....	366
Extra Credit.....	369
Further Reading.....	370
Seeking the Proper Level.....	371
Step-By-Step Instructions.....	371
Step #1: Define an Interface for Mobile Numbers.....	371
Step #2: Implement the Interface: the New Way.....	372

Step #3: Implement the Interface: the Old Way.....	373
Step #4: Choose and Use the Bridge.....	374
Extra Credit.....	375
Further Reading.....	375
Appendix A: How To Get Started.....	377
Java.....	377
Step #1: Install the JDK.....	378
Step #2: Learn Java.....	378
Install the Android SDK.....	379
Step #1: Install the Base Tools.....	379
Step #2: Install the SDKs and Add-Ons.....	379
Install the ADT for Eclipse.....	383
Install Apache Ant.....	385
Set Up the Emulator.....	386
Set Up the Device.....	393
Step #1: Windows.....	394
Step #2: OS X and Linux.....	395
Appendix B: Coping with Eclipse.....	397
How to Import a Non-Eclipse Project.....	397
How to Get To DDMS.....	402
How to Create an Emulator.....	404
How to Run a Project.....	405
How Not to Run Your Project.....	406
How to Get Past Eclipse.....	406

Welcome to the Warescription!

We hope you enjoy this ebook and its updates – subscribe to the Warescription newsletter on the [Warescription](#) site to learn when new editions of this book, or other books, are available.

All editions of CommonsWare titles, print and ebook, follow a software-style numbering system. Major releases (1.0, 2.0, etc.) are available in both print and ebook; minor releases (0.1, 0.9, etc.) are available in ebook form for Warescription subscribers only. Releases ending in .9 are "release candidates" for the next major release, lacking perhaps an index but otherwise being complete.

Each Warescription ebook is licensed for the exclusive use of its subscriber and is tagged with the subscriber's name. We ask that you not distribute these books. If you work for a firm and wish to have several employees have access, enterprise Warescriptions are available. Just contact us at enterprise@commonsware.com.

Also, bear in mind that eventually this edition of this title will be released under a Creative Commons license – more on this in the [preface](#).

Remember that the CommonsWare Web site has errata and resources (e.g., source code) for each of our titles. Just visit the Web page for the book you are interested in and follow the links.

You can search through the PDF using most PDF readers (e.g., Adobe Reader). If you wish to search all of the CommonsWare books at once, and

your operating system does not support that directly, you can always combine the PDFs into one, using tools like [PDF Split-And-Merge](#) or the Linux command `pdftk *.pdf cat output combined.pdf`.

Welcome to the Book!

If you come to this book after having read its companion volumes, *The Busy Coder's Guide to Android Development* and *The Busy Coder's Guide to Advanced Android Development*, thanks for sticking with the series! CommonsWare aims to have the most comprehensive set of Android development resources (outside of the Open Handset Alliance itself), and we appreciate your interest.

If you come to this book having learned about Android from other sources, thanks for joining the CommonsWare community!

Prerequisites

This book is a collection of tutorials, walking you through developing Android applications, from the simplest "Hello, world!" to applications using many advanced Android APIs.

Since this book only supplies tutorials, **you will want something beyond it as a reference guide**. That could be simply the Android SDK documentation, available with your SDK installation or online. It could be the other books in the CommonsWare Android series. Or, it could be another Android book – a list of currently-available Android books can be found on the [Android Programming knol](#). What you do not want to do is

attempt to learn all of Android solely from these tutorials, as they will demonstrate the breadth of the Android API but not its depth.

Also, the tutorials themselves have varying depth. Early on, there is more "hand-holding" to explain every bit of what needs to be done (e.g., classes to import). As the tutorials progress, some of the simpler Java bookkeeping steps are left out of the instructions – such as exhaustive lists of import statements – so the tutorials can focus on the Android aspects of the code.

You can find out when new releases of this book are available via:

- The [cw-android](#) Google Group, which is also a great place to ask questions about the book and its examples
- The [commonsguy](#) Twitter feed
- The [CommonsBlog](#)
- The Warescription newsletter, which you can subscribe to off of your [Warescription](#) page

Using the Tutorials

Each tutorial has a main set of step-by-step instructions, plus an "Extra Credit" section. The step-by-step instructions are intended to guide you through creating or extending Android applications, including all code you need to enter and all commands you need to run. The "Extra Credit" sections, on the other hand, provide some suggested areas for experimentation beyond the base tutorial, without step-by-step instructions.

If you wish to start somewhere in the middle of the book, or if you only wish to do the "Extra Credit" work, or if you just want to examine the results without doing the tutorials directly yourself, you can download the results of each tutorial's step-by-step instructions [from the book's github repository](#). You can either clone the repository, or click the Download Source button in the upper-right to get the source as a ZIP file. The source code is organized by tutorial number, so you can readily find the project(s) associated with a particular tutorial from the book.

Note that while you are welcome to copy and paste code out of the book, you may wish to copy from [the full source code](#) instead. A side-effect of the way the source code listings are put into this book makes them difficult to copy from some PDF viewers, for example.

The tutorials do not assume you are using Eclipse, let alone any other specific editor or debugger. The instructions included in the tutorials will speak in general terms when it comes to tools outside of those supplied by the Android SDK itself.

The code for the tutorials has been tested most recently on Android 2.2. It should work on older versions as well, on the whole.

The tutorials include instructions for both Linux and Windows XP. OS X developers should be able to follow the Linux instructions in general, making slight alterations as needed for your platform. Windows Vista users should be able to follow the Windows XP instructions in general, tweaking the steps to deal with Vista's directory structure and revised Start menu.

If you wish to use the source code from the CommonsWare Web site, bear in mind a few things:

1. The projects are set up to be built by Ant, not by Eclipse. If you wish to use the code with Eclipse, you will need to create a suitable Android Eclipse project and import the code and other assets.
2. You should delete build.xml, then run `android update project -p ...` (where ... is the path to a project of interest) on those projects you wish to use, so the build files are updated for your Android SDK version.

Also, please note that the tutorials are set up to work well on HVGA and larger screen sizes. Using them on QVGA or similar sizes is not recommended.

Warescription

This book will be published both in print and in digital form. The digital versions of all CommonsWare titles are available via an annual subscription – the Warescription.

The Warescription entitles you, for the duration of your subscription, to digital forms of *all* CommonsWare titles, not just the one you are reading. Presently, CommonsWare offers PDF and Kindle; other digital formats will be added based on interest and the openness of the format.

Each subscriber gets personalized editions of all editions of each title: both those mirroring printed editions and in-between updates that are only available in digital form. That way, your digital books are never out of date for long, and you can take advantage of new material as it is made available instead of having to wait for a whole new print edition. For example, when new releases of the Android SDK are made available, this book will be quickly updated to be accurate with changes in the APIs.

From time to time, subscribers will also receive access to subscriber-only online material, including not-yet-published new titles.

Also, if you own a print copy of a CommonsWare book, and it is in good clean condition with no marks or stickers, you can [exchange that copy](#) for a free four-month Warescription.

If you are interested in a Warescription, visit the Warescription section of the CommonsWare [Web site](#).

What's New

For those of you who have a Warescription, or otherwise have been keeping up with this book, here is what is new in this version:

- The Patchy examples were tweaked to use identi.ca instead of Twitter, due to the latter's change in authentication schemes
- The tutorials were tested on Android 2.2

About the "Further Reading" Sections

Each tutorial has, at the end, a section named "Further Reading". Here, we list places to go learn more about the theory behind the techniques illustrated in the preceding tutorial. Bear in mind, however, that the Internet is fluid, so links may not necessarily work. And, of course, there is no good way to link to other books. Hence, the "Further Reading" section describes where you can find material, but actually getting there may require a few additional clicks on your part. We apologize for the inconvenience.

Errata and Book Bug Bounty

Books updated as frequently as CommonsWare's inevitably have bugs. Flaws. Errors. Even the occasional gaffe, just to keep things interesting. You will find a list of the known bugs on the [errata page](#) on the CommonsWare Web site.

But, there are probably even more problems. If you find one, please let us know!

Be the first to report a unique concrete problem in the current digital edition, and we'll give you a coupon for a six-month Warescription as a bounty for helping us deliver a better product. You can use that coupon to get a new Warescription, renew an existing Warescription, or give the coupon to a friend, colleague, or some random person you meet on the subway.

By "concrete" problem, we mean things like:

- Typographical errors
- Sample applications that do not work as advertised, in the environment described in the book
- Factual errors that cannot be open to interpretation

By "unique", we mean ones not yet reported. Each book has an errata page on the CommonsWare Web site; most known problems will be listed there. One coupon is given per email containing valid bug reports.

NOTE: Books with version numbers lower than 0.9 are ineligible for the bounty program, as they are in various stages of completion. We appreciate bug reports, though, if you choose to share them with us.

We appreciate hearing about "softer" issues as well, such as:

- Places where you think we are in error, but where we feel our interpretation is reasonable
- Places where you think we could add sample applications, or expand upon the existing material
- Samples that do not work due to "shifting sands" of the underlying environment (e.g., changed APIs with new releases of an SDK)

However, those "softer" issues do not qualify for the formal bounty program.

Be sure to check [the book's errata page](#), though, to see if your issue has already been reported.

Questions about the bug bounty, or problems you wish to report for bounty consideration, should be sent to [CommonsWare](#).

Source Code License

The source code samples shown in this book are available for download from the [book's GitHub repository](#). All of the Android projects are licensed under the [Apache 2.0 License](#), in case you have the desire to reuse any of it.

Creative Commons and the Four-to-Free (42F) Guarantee

Each CommonsWare book edition will be available for use under the [Creative Commons Attribution-Noncommercial-ShareAlike 3.0](#) license as of the fourth anniversary of its publication date, or when 4,000 copies of the edition have been sold, whichever comes first. That means that, once four years have elapsed (perhaps sooner!), you can use this prose for non-commercial purposes. That is our Four-to-Free Guarantee to our readers

and the broader community. For the purposes of this guarantee, new Warescriptions and renewals will be counted as sales of this edition, starting from the time the edition is published.

This edition of this book will be available under the aforementioned Creative Commons license on January 1, 2015. Of course, watch the CommonsWare Web site, as this edition might be relicensed sooner based on sales.

For more details on the Creative Commons Attribution-Noncommercial-ShareAlike 3.0 license, visit the Creative Commons Web site.

Note that future editions of this book will become free on later dates, each four years from the publication of that edition or based on sales of that specific edition. Releasing one edition under the Creative Commons license does not automatically release *all* editions under that license.

Lifecycle of a CommonsWare Book

CommonsWare books generally go through a series of stages.

First are the pre-release editions. These will have version numbers below 0.9 (e.g., 0.2). These editions are incomplete, often times having but a few chapters to go along with outlines and notes. However, we make them available to those on the Warescription so they can get early access to the material.

Release candidates are editions with version numbers ending in ".9" (0.9, 1.9, etc.). These editions should be complete. Once again, they are made available to those on the Warescription so they get early access to the material and can file bug reports (and receive bounties in return!).

Major editions are those with version numbers ending in ".0" (1.0, 2.0, etc.). These will be first published digitally for the Warescription members, but will shortly thereafter be available in print from booksellers worldwide.

Versions between a major edition and the next release candidate (e.g., 1.1, 1.2) will contain bug fixes plus new material. Each of these editions should also be complete, in that you will not see any "TBD" (to be done) markers or the like. However, these editions may have bugs, and so bug reports are eligible for the bounty program, as with release candidates and major releases.

A book usually will progress fairly rapidly through the pre-release editions to the first release candidate and Version 1.0 – often times, only a few months. Depending on the book's scope, it may go through another cycle of significant improvement (versions 1.1 through 2.0), though this may take several months to a year or more. Eventually, though, the book will go into more of a "maintenance mode", only getting updates to fix bugs and deal with major ecosystem events – for example, a new release of the Android SDK will necessitate an update to all Android books.

Roster of Tutorials

Here is what you can expect in going through the tutorials in this book:

1. We start off with a simple throwaway project, just to make sure you have the development tools all set up properly.
2. We then begin creating `LunchList`, an application to track restaurants where you might wish to go for lunch. In this tutorial, we set up a simple form to collect basic information about a restaurant, such as a name and address.
3. We expand the form to add radio buttons for the type of restaurant (e.g., takeout).
4. Instead of tracking just a single restaurant, we add support for a list of restaurants – but each restaurant shows up in the list only showing its name.
5. We extend the list to show the name and address of each restaurant, plus an icon for the restaurant type.
6. To give us more room, we split the UI into two tabs, one for the list of restaurants, and one for the detail form for a restaurant.

7. We experiment with an option menu (the kind that appears when you press the MENU button on a phone) and display a pop-up message.
8. We learn how to start a background thread and coordinate communications between the background thread and the main ("UI") thread.
9. We learn how to find out when the activity is going off-screen, stopping and restarting our background thread as needed.
10. We create a separate UI description for what the tabs should look like when the phone is held in a landscape orientation.
11. We finally add database support, so your restaurant data persists from run to run of the application.
12. We eliminate the tabs and split the UI into two separate screens ("activities"), one for the list of restaurants, and one for the detail form to add or edit a restaurant.
13. We establish a shared preference – and an activity to configure it – to allow the user to specify the sort order of the restaurants in the list.
14. We re-establish the landscape version of our UI (lost when we eliminated the tabs in Tutorial 12) and experiment with how to handle the orientation changing during execution of our application.
15. We put `LunchList` on hold and start up a brand new project, `Patchy`, for accessing identi.ca.
16. We integrate the JTtwitter JAR – an open source Java API for Twitter and things supporting the Twitter API – into our application.
17. We add a partial implementation of a service to the application, one that will periodically poll identi.ca for timeline updates.
18. We fully integrate the service from Tutorial 17 into the application, showing the timeline updates in a list in the main activity.
19. We split the service out into a separate project, accessed as a "remote service" using inter-process communication (IPC).

20. We add logic to the service to watch for posts from specific people – our "BFFs" – and display an icon in the status bar when we see such a post.
21. We add a menu choice to allow the user to inject their current location (latitude and longitude) into the status update they are editing.
22. We watch for locations embedded in posts – akin to those created in Tutorial 21 – and, if the user clicks on a post in the timeline with such a location, we display that location on a map.
23. We add a "helpcast" video to our application, to demonstrate video playback integration.
24. We supplement the "helpcast" with a more traditional help page, in HTML format, show in an embedded WebKit browser.
25. We extend the help page to pull our identi.ca screen name in for customization, by building a bridge between the Javascript in the Web page and the Android Java environment.
26. We give the user a menu option to hide and show the widgets associated with updating the identi.ca status, then arrange to animate hiding and showing those widgets.
27. We replace the callback system used previously in our application with one that uses broadcast Intent objects.
28. We integrate our application into the contacts engine, so posts from people who also our in our contacts database are highlighted.
29. Back in `Patchy`, we start monitoring for when Android says the battery level changes, so when it gets low enough, we scale back the frequency of our polls for timeline updates.
30. We have `Patchy`'s service monitor for timeline updates via a scheduled "alarm" rather than via a background thread sleeping for a certain amount of time.
31. We tie `LunchList`'s restaurant database into the Android search framework, so you can find a restaurant by name.
32. We create another project, one that will allow us to pick a contact out of the contact database, then find out what activities can be launched that know how to do something with that contact.

33. We create a simple app widget (i.e., interactive element for the home screen) for `LunchList`, one that displays the name of a randomly-selected restaurant.
34. We extend the app widget to allow the user to click a button to choose a different restaurant and click the name of the restaurant to pull up the `LunchList` detail form for that restaurant.
35. We test the `LunchList` application using the Test Monkey, simulating random input into the UI.
36. We allow users to provide phone numbers for restaurants, then add in phone-dialing capability to `LunchList`.
37. We (pretend to) allow users to take pictures of restaurants using the device's built-in camera.
38. We wrap up by allowing users to randomly select a restaurant by shaking their phone while running the `LunchList` application.
39. We allow users to send the name and address of a restaurant to others via SMS.
40. We fix up the SMS code to support both Android 1.x and Android 2.x.

PART I – Introductory Tutorials

Your First Android Project

There are two major steps for getting started with Android:

1. You need to install the Android SDK and developer tools
2. You should build a test project to confirm that those tools are properly installed and configured

If you have already done some form of "hello, world" project with the development tools on your development machine, you can skip this tutorial.

If you have not yet installed the Android SDK and related tools, there is an [appendix](#) that covers this process. Once you have the Android SDK, it is time to make your first Android project. The good news is that this requires zero lines of code – Android's tools create a "Hello, world!" application for you as part of creating a new project. All you need to do is build it, install it, and see it come up on your emulator or device. That is what this tutorial is for.

Step #1: Create the New Project

Android's tools can create a complete skeleton project for you, with everything you need for a complete (albeit very trivial) Android application. The only real difference comes from whether you are using Eclipse or the command line.

Step #1: Eclipse

From the Eclipse main menu, choose File | New | Project..., and this will bring up a list of project types to choose from. Fold open the Android option and click on Android Project:

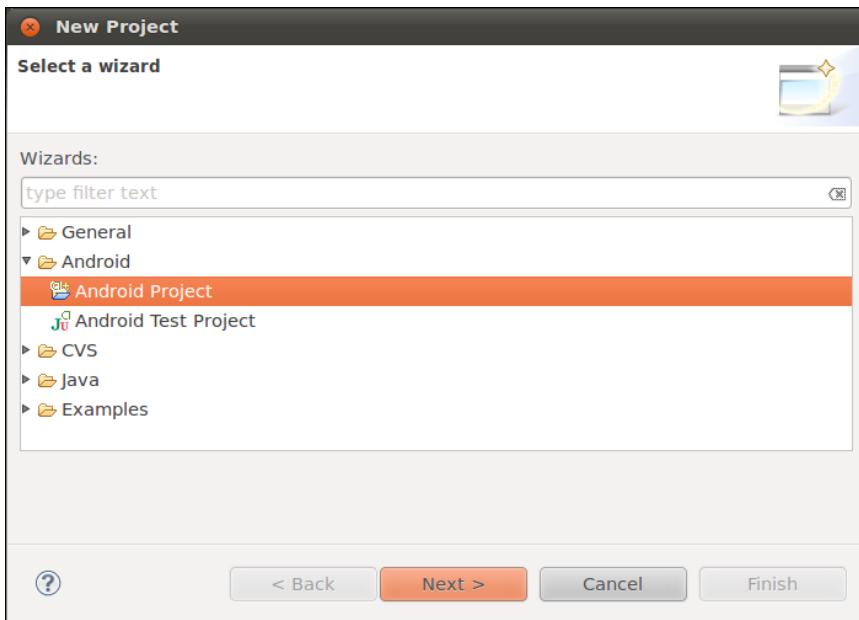


Figure 1. Eclipse New Project Wizard

Press Next to advance the wizard to the main Android project page:

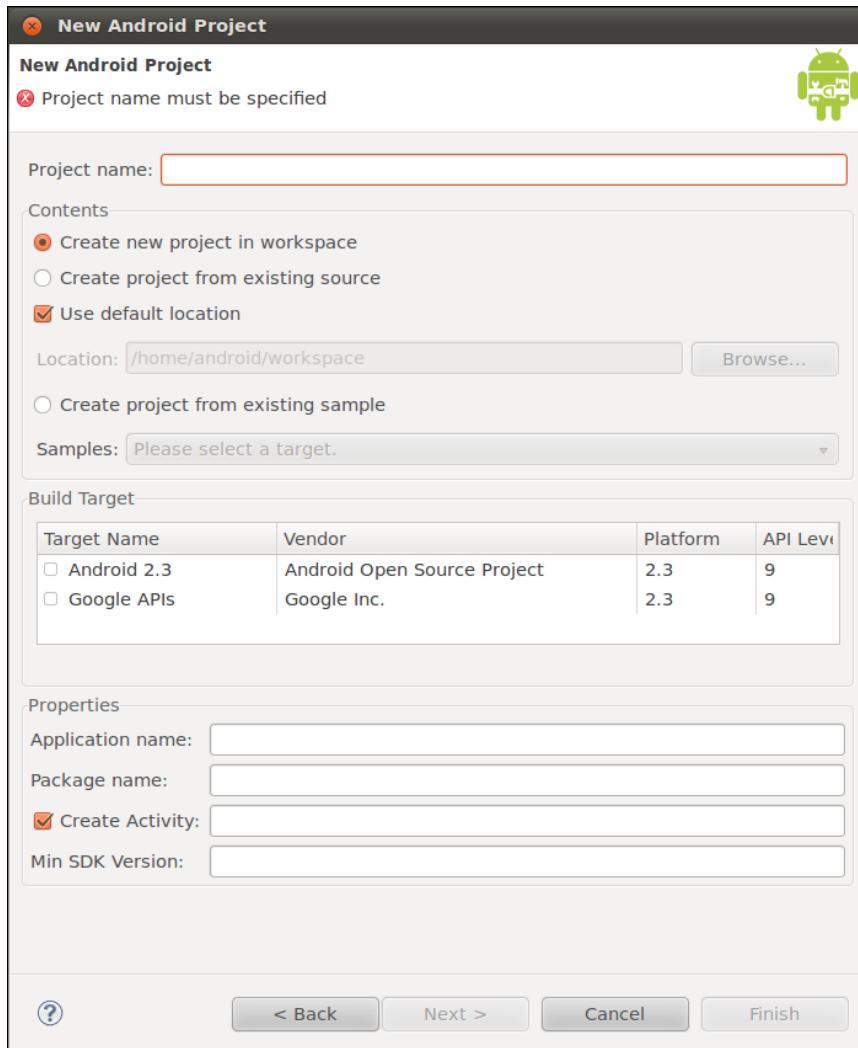


Figure 2. Eclipse New Project Wizard, Android Project

Fill in the following:

- The name of the project (e.g., Now)
- The Android SDK you wish to compile against (e.g., Google APIs for Android 2.3)
- The name of the Java package in which this project goes (e.g., com.commonsware.android.skeleton)

Your First Android Project

- The name of the initial activity to create (e.g., Now)

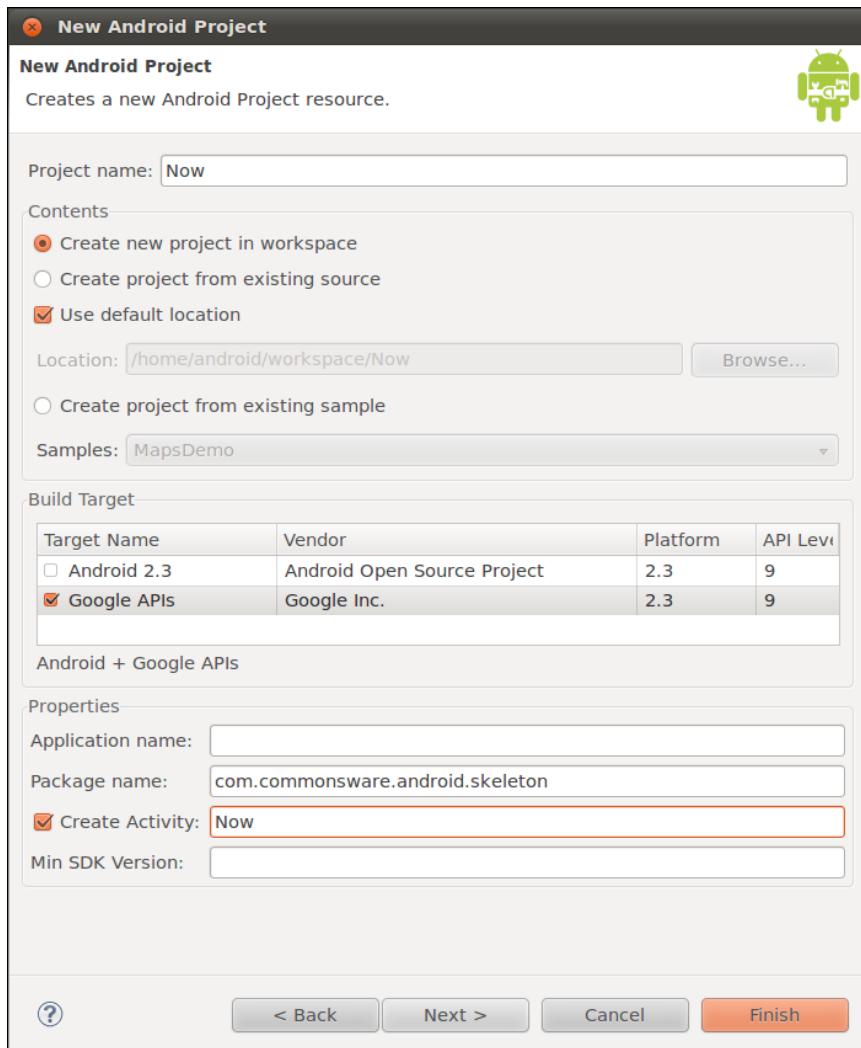


Figure 3. Eclipse New Project Wizard, Android Project (continued)

At this point, clicking Finish will create your Eclipse project.

Step #2: Command Line

Here is a sample command that creates an Android project from the command line:

```
android create project --target "Google Inc.:Google APIs:7" --path Skeleton/Now  
--activity Now --package com.commonsware.android.skeleton
```

This will create an application skeleton for you, complete with everything you need to build your first Android application: Java source code, build instructions, etc. However, you are probably going to need to customize this somewhat. Here are what those command-line switches mean:

- `--target` indicates what version of Android you are "targeting" in terms of your build process. You need to supply the ID of a target that is installed on your development machine, one you downloaded via the SDK and AVD Manager. You can find out what targets are available via the `android list targets` command. Typically, your build process will target the newest version of Android that you have available.
- `--path` indicates where you want the project files to be generated. Android will create a directory if the one you name does not exist. For example, in the command shown above, a `Skeleton/Now/` directory will be created (or used if it exists) underneath the current working directory, and the project files will be stored there.
- `--activity` indicates the Java class name of your first activity for this project. Do not include a package name, and the name has to meet Java class naming conventions.
- `--package` indicates the Java package in which your first activity will be located. This package also uniquely identifies your project on any device on which you install it, and this package also needs to be unique on the Android Market if you plan on distributing your application there. Hence, typically, you construct your package based on a domain name you own (e.g., `com.commonsware.android.skeleton`), to reduce the odds of an accidental package name collision with somebody else.

For your development machine, you will need to pick a suitable target, and you may wish to change the path. The activity and package you can leave alone for now.

Step #2: Build, Install, and Run the Application in Your Emulator or Device

Having a project is nice and all, but it would be even better if we could build and run it, whether on the Android emulator or your Android device. Once again, the process differs somewhat depending on whether you are using Eclipse or not.

Step #1: Eclipse

With your project selected in the Package Explorer pane, click the green "play" button in the Eclipse toolbar to run your project. The first time you do this, you will have to go through a few steps to set up a "run configuration", so Eclipse knows what you want to do.

First, in the "Run As" list, choose "Android Application":

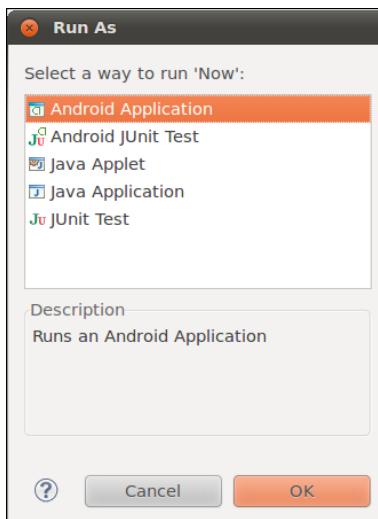


Figure 4. Eclipse "Run As" List

If you have more than one emulator AVD or device available, you will then get an option to choose which you wish to run the application on. Otherwise, if you do not have a device plugged in, the emulator will start up with the AVD you created earlier. Then, Eclipse will install the application on your device or emulator and start it up.

Step #2: Command Line

For developers not using Eclipse, in your terminal, change into the `Skeleton/Now` directory, then run the following command:

```
ant clean install
```

The Ant-based build should emit a list of steps involved in the installation process, which look like this:

```
Buildfile: /home/some-balding-guy/projects/Skeleton/Now/build.xml
[setup] Android SDK Tools Revision 8
[setup] Project Target: Google APIs
[setup] Vendor: Google Inc.
[setup] Platform Version: 2.1-update1
[setup] API level: 7
[setup]
[setup] -----
[setup] Resolving library dependencies:
[setup] No library dependencies.
[setup]
[setup] -----
[setup]
[setup] WARNING: No minSdkVersion value set. Application will install on all
Android versions.
[setup]
[setup] Importing rules file: tools/ant/main_rules.xml

clean:
[delete] Deleting directory /home/some-balding-guy/projects/Skeleton/Now/bin

-debug-obfuscation-check:

-set-debug-mode:

-compile-tested-if-test:

-dirs:
[echo] Creating output directories if needed...
[mkdir] Created dir: /home/some-balding-guy/projects/Skeleton/Now/bin
```

```
[mkdir] Created dir: /home/some-balding-guy/projects/Skeleton/Now/gen
[mkdir] Created dir: /home/some-balding-guy/projects/Skeleton/Now/bin/classes

-pre-build:

-resource-src:
[echo] Generating R.java / Manifest.java from the resources...

-aidl:
[echo] Compiling aidl files into Java classes...

-pre-compile:

compile:
[javac] /opt/android-sdk-linux/tools/ant/main_rules.xml:361: warning:
'includeantruntime' was not set, defaulting to build.sysclasspath=last; set to
false for repeatable builds
[javac] Compiling 2 source files to /home/some-balding-
guy/projects/Skeleton/Now/bin/classes

-post-compile:

-obfuscate:

-dex:
[echo] Converting compiled files and external libraries into /home/some-
balding-guy/projects/Skeleton/Now/bin/classes.dex...

-package-resources:
[echo] Packaging resources
[aapt] Creating full resource package...

-package-debug-sign:
[apkbuilder] Creating Now-debug-unaligned.apk and signing it with a debug key...

debug:
[echo] Running zip align on final apk...
[echo] Debug Package: /home/some-balding-guy/projects/Skeleton/Now/bin/Now-
debug.apk

BUILD SUCCESSFUL
Total time: 4 seconds
```

Note the `BUILD SUCCESSFUL` at the bottom – that is how you know the application compiled successfully.

When you have a clean build, in your emulator or device, open up the application launcher, typically found at the bottom of the home screen:



Figure 5. Android emulator application launcher

Notice there is an icon for your Now application. Click on it to open it and see your first activity in action. To leave the application and return to the launcher, press the "BACK button", located to the right of the [MENU] button, and looks like an arrow pointing to the left.

A Simple Form

This tutorial is the first of several that will build up a "lunch list" application, where you can track various likely places to go to lunch. While this application may seem silly, it will give you a chance to exercise many features of the Android platform. Besides, perhaps you may even find the application to be useful someday.

Step-By-Step Instructions

Here is how you can create this application:

Step #1: Generate the Application Skeleton

First, we need to create a new project.

Eclipse

Use the new-project wizard to create an empty Android project named `LunchList`, as described in the [Android developer documentation](#). This will create an application skeleton for you, complete with everything you need to build your first Android application: Java source code, build instructions, etc.

Outside of Eclipse

Inside your terminal (e.g., Command Prompt for Windows), switch to a directory in which you would like the project to reside. Then, run the following command:

```
android create project --target "Google Inc.:Google APIs:6" --path ./LunchList  
--activity LunchList --package apt.tutorial
```

This will create an application skeleton for you, complete with everything you need to start building the `LunchList` application.

Step #2: Modify the Layout

Using your text editor, open the `LunchList/res/layout/main.xml` file. Initially, that file will look like this:

```
<?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:text="Hello World, LunchList"  
    />  
</LinearLayout>
```

Change that layout to look like this:

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

```
    android:layout_height="wrap_content"
    android:text="Name:"
  />
<EditText android:id="@+id/name"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  />
</LinearLayout>
<LinearLayout
  android:orientation="horizontal"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
>
<TextView
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Address:"
/>
<EditText android:id="@+id/addr"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  />
</LinearLayout>
<Button android:id="@+id/save"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:text="Save"
/>
</LinearLayout>
```

This gives us a three-row form: one row with a labeled field for the restaurant name, one with a labeled field for the restaurant address, and a big Save button.

Step #3: Compile and Install the Application

Compile and install the application in the emulator by running the following commands in your terminal:

```
ant clean install
```

Or, from Eclipse, just run the project.

Step #4: Run the Application in the Emulator

In your emulator, in the application launcher, you will see an icon for your LunchList application. Click it to bring up your form:

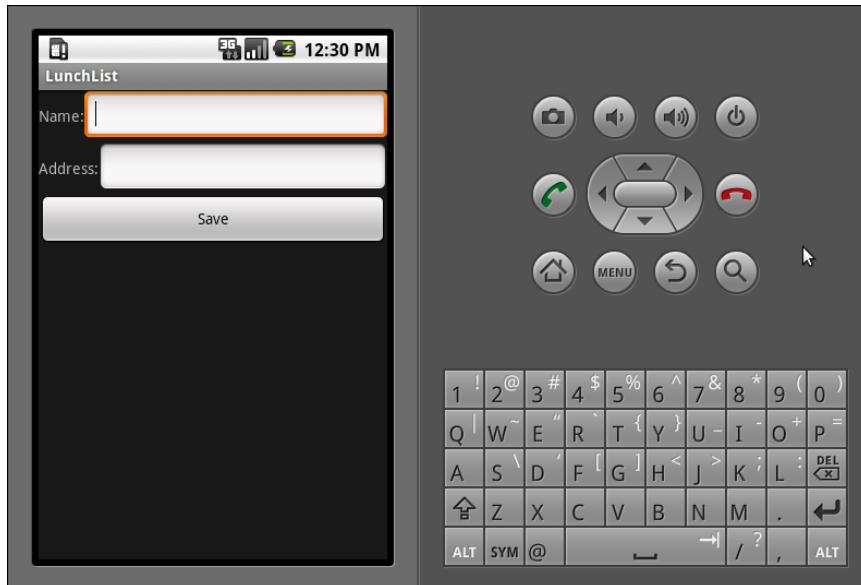


Figure 6. The first edition of LunchList

Use the directional pad (D-pad) below the [MENU] button to navigate between the fields and button. Enter some text in the fields and click the button, to see how those widgets behave. Then, click the BACK button to return to the application launcher.

Step #5: Create a Model Class

Now, we want to add a class to the project that will hold onto individual restaurants that will appear in the LunchList. Right now, we can only really work with one restaurant, but that will change in a future tutorial.

So, using your text editor, create a new file named `LunchList/src/apt/tutorial/Restaurant.java` with the following contents:

```
package apt.tutorial;

public class Restaurant {
    private String name="";
    private String address="";

    public String getName() {
        return(name);
    }

    public void setName(String name) {
        this.name=name;
    }

    public String getAddress() {
        return(address);
    }

    public void setAddress(String address) {
        this.address=address;
    }
}
```

This is simply a rudimentary model, with private data members for the name and address, and getters and setters for each of those.

Of course, don't forget to save your changes!

Step #6: Save the Form to the Model

Finally, we want to hook up the Save button, such that when it is pressed, we update a restaurant object based on the two `EditText` fields. To do this, open up the `LunchList/src/apt/tutorial/LunchList.java` file and replace the generated `Activity` implementation with the one shown below:

```
package apt.tutorial;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;

public class LunchList extends Activity {
    Restaurant r=new Restaurant();

    @Override
```

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

    Button save=(Button)findViewById(R.id.save);

    save.setOnClickListener(onSave);
}

private View.OnClickListener onSave=new View.OnClickListener() {
    public void onClick(View v) {
        EditText name=(EditText)findViewById(R.id.name);
        EditText address=(EditText)findViewById(R.id.addr);

        r.setName(name.getText().toString());
        r.setAddress(address.getText().toString());
    }
};
```

Here, we:

- Create a single local restaurant instance when the activity is instantiated
- Get our Button from the Activity via `findViewById()`, then connect it to a listener to be notified when the button is clicked
- In the listener, we get our two `EditText` widgets via `findViewById()`, then retrieve their contents and put them in the restaurant

This code sample shows the use of an anonymous inner class implementation of a `View.OnClickListener`, named `onSave`. This technique is used in many places throughout this book, as it is a convenient way to organize bits of custom code that go into these various listener objects.

Then, run the `ant install` command to compile and update the emulator. Run the application to make sure it seems like it runs without errors, though at this point we are not really using the data saved in the restaurant object just yet.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Instead of using the console tools as documented above, try using Eclipse. You will need to download Eclipse, install the Android plug-in, and use it to create your first project.
- Try replacing the icon for your application. To do this, you will need to find a suitable 48x48 pixel image, create a `drawable/` directory inside your `res/` directory in the project, and adjust the `AndroidManifest.xml` file to contain an `android:icon = "@drawable/my_icon"` attribute in the `application` element, where `my_icon` is replaced by the base name of your image file.
- Try playing with the fonts for use in both the `TextView` and `EditText` widgets. The Android SDK documentation will show a number of XML attributes you can manipulate to change the color, make the text boldface, etc.

Further Reading

You can learn more about XML layouts in the "Using XML-Based Layouts" chapter of [The Busy Coder's Guide to Android Development](#). Similarly, you can learn more about simple widgets, like fields and buttons, in the "Employing Basic Widgets" chapter of the same book, where you will also find "Working with Containers" for container classes like `LinearLayout`.

A Fancier Form

In this tutorial, we will switch to using a `TableLayout` for our restaurant data entry form, plus add a set of radio buttons to represent the type of restaurant.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the `02-SimpleForm` edition of `LunchList` to use as a starting point.

Step #1: Switch to a `TableLayout`

First, open `LunchList/res/layout/main.xml` and modify its contents to look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:stretchColumns="1"
    >
    <TableRow>
        <TextView android:text="Name:>
        <EditText android:id="@+id/name" />
    </TableRow>
    <TableRow>
```

```
<TextView android:text="Address:" />
<EditText android:id="@+id/addr" />
</TableRow>
<Button android:id="@+id/save"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Save"
/>
</TableLayout>
```

Notice that we replaced the three `LinearLayout` containers with a `TableLayout` and two `TableRow` containers. We also set up the `EditText` column to be stretchable.

Recompile and reinstall the application, then run it in the emulator. You should see something like this:

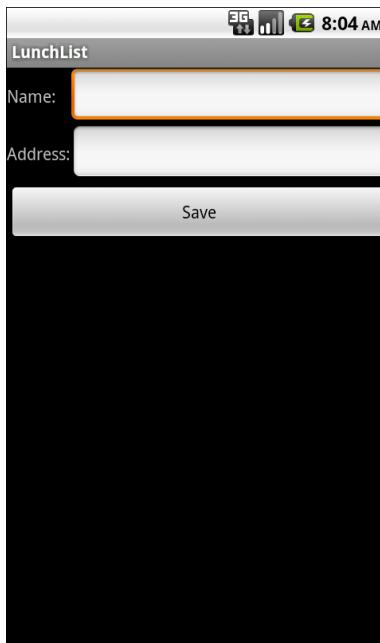


Figure 7. Using a TableLayout

Notice how the two `EditText` fields line up, whereas before, they appeared immediately after each label.

NOTE: At this step, or any other, when you try to run your application, you may get the following screen:



Figure 8. A "force-close" dialog

If you encounter this, first try to do a full rebuild of the project. In Eclipse, this would involve doing **Project > Force Clean**. At the command line, use `ant clean` or delete the contents of your `bin/` and `gen/` directories, then `ant install`. If the problem persists after this, then there is a bug in your code somewhere. You can use `adb logcat`, DDMS, or the DDMS perspective in Eclipse to see the Java stack trace associated with this crash, to help you perhaps diagnose what is going on.

Step #2: Add a RadioGroup

Next, we should add some `RadioButton` widgets to indicate the type of restaurant this is: one that offers take-out, one where we can sit down, or one that is only a delivery service.

To do this, modify `LunchList/res/layout/main.xml` once again, this time to look like:

A Fancier Form

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:stretchColumns="1"
    >
    <TableRow>
        <TextView android:text="Name:" />
        <EditText android:id="@+id/name" />
    </TableRow>
    <TableRow>
        <TextView android:text="Address:" />
        <EditText android:id="@+id/addr" />
    </TableRow>
    <TableRow>
        <TextView android:text="Type:" />
        <RadioGroup android:id="@+id/types">
            <RadioButton android:id="@+id/take_out"
                android:text="Take-Out"
            />
            <RadioButton android:id="@+id/sit_down"
                android:text="Sit-Down"
            />
            <RadioButton android:id="@+id/delivery"
                android:text="Delivery"
            />
        </RadioGroup>
    </TableRow>
    <Button android:id="@+id/save"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Save"
    />
</TableLayout>
```

Our RadioGroup and RadioButton widgets go inside the TableLayout, so they will line up with the rest of table – you can see this once you recompile, reinstall, and run the application:

A Fancier Form

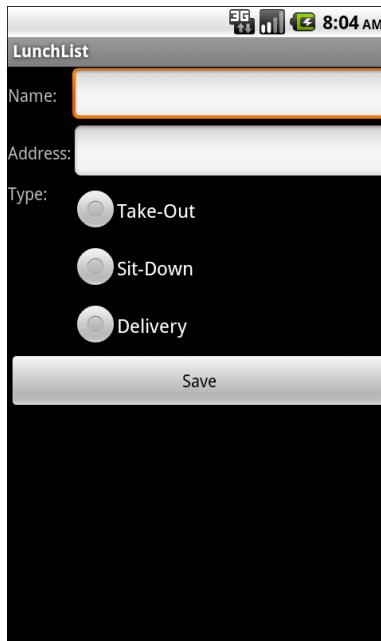


Figure 9. Adding radio buttons

Step #3: Update the Model

Right now, our model class has no place to hold the restaurant type. To change that, modify `LunchList/src/apt/tutorial/Restaurant.java` to add in a new `private String type` data member and a getter/setter pair, like these:

```
public String getType() {
    return(type);
}

public void setType(String type) {
    this.type=type;
}
```

When you are done, your restaurant class should look something like this:

```
package apt.tutorial;

public class Restaurant {
    private String name="";
    private String address="";
```

```
private String type="";  
  
public String getName() {  
    return(name);  
}  
  
public void setName(String name) {  
    this.name=name;  
}  
  
public String getAddress() {  
    return(address);  
}  
  
public void setAddress(String address) {  
    this.address=address;  
}  
  
public String getType() {  
    return(type);  
}  
  
public void setType(String type) {  
    this.type=type;  
}
```

Step #4: Save the Type to the Model

Finally, we need to wire our `RadioButton` widgets to the model, such that when the user clicks the Save button, the type is saved as well. To do this, modify the `onSave` listener object to look like this:

```
private View.OnClickListener onSave=new View.OnClickListener() {  
    public void onClick(View v) {  
        EditText name=(EditText)findViewById(R.id.name);  
        EditText address=(EditText)findViewById(R.id.addr);  
  
        r.setName(name.getText().toString());  
        r.setAddress(address.getText().toString());  
  
        RadioGroup types=(RadioGroup)findViewById(R.id.types);  
  
        switch (types.getCheckedRadioButtonId()) {  
            case R.id.sit_down:  
                r.setType("sit_down");  
                break;  
  
            case R.id.take_out:  
                r.setType("take_out");  
        }  
    }  
}
```

```
        break;

    case R.id.delivery:
        r.setType("delivery");
        break;
    }
};

};
```

Note that you will also need to import android.widget.RadioGroup for this to compile. The full activity will then look like this:

```
package apt.tutorial;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.RadioGroup;

public class LunchList extends Activity {
    Restaurant r=new Restaurant();

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

        Button save=(Button)findViewById(R.id.save);

        save.setOnClickListener(onSave);
    }

    private View.OnClickListener onSave=new View.OnClickListener() {
        public void onClick(View v) {
            EditText name=(EditText)findViewById(R.id.name);
            EditText address=(EditText)findViewById(R.id.addr);

            r.setName(name.getText().toString());
            r.setAddress(address.getText().toString());

            RadioGroup types=(RadioGroup)findViewById(R.id.types);

            switch (types.getCheckedRadioButtonId()) {
                case R.id.sit_down:
                    r.setType("sit_down");
                    break;

                case R.id.take_out:
                    r.setType("take_out");
                    break;
            }
        }
    };
}
```

```
        case R.id.delivery:  
            r.setType("delivery");  
            break;  
        }  
    };  
}
```

Recompile, reinstall, and run the application. Confirm that you can save the restaurant data without errors.

If you are wondering what will happen if there is no selected `RadioButton`, the `RadioGroup` call to `getCheckedRadioButtonId()` will return -1, which will not match anything in our `switch` statement, and so the model will not be modified.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- If you have an Android device, try installing the app on the device and running it there. The easiest way to do this is to shut down your emulator, plug in your device, and run `ant reinstall`.
- Set one of the three radio buttons to be selected by default, using `android:checked = "true"`.
- Try creating the `RadioButton` widgets in Java code, instead of in the layout. To do this, you will need to create the `RadioButton` objects themselves, configure them (e.g., supply them with text to display), then add them to the `RadioGroup` via `addView()`.
- Try adding more `RadioButton` widgets than there are room to display on the screen. Note how the screen does not automatically scroll to show them. Then, wrap your entire layout in a `ScrollView` container, and see how the form can now scroll to accommodate all of your widgets.

Further Reading

You can learn more about radio buttons in the "Employing Basic Widgets" chapter of [The Busy Coder's Guide to Android Development](#). Also, you will find material on `TableLayout` in the "Working with Containers" chapter of the same book.

Adding a List

In this tutorial, we will change our model to be a list of restaurants, rather than just one. Then, we will add a `ListView` to view the available restaurants. This will be rather incomplete, in that we can only add a new restaurant, not edit or delete an existing one, but we will cover those steps too in a later tutorial.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the `03-FancierForm` edition of `LunchList` to use as a starting point.

Step #1: Hold a List of Restaurants

First, if we are going to have a list of restaurants in the UI, we need a list of restaurants as our model. So, in `LunchList`, change:

```
Restaurant r=new Restaurant();
```

to:

```
List<Restaurant> model=new ArrayList<Restaurant>();
```

Note that you will need to import `java.util.List` and `java.util.ArrayList` as well.

Step #2: Save Adds to List

Note that the above code will not compile, because our `onSave` button click handler is still set up to reference the old single restaurant model. For the time being, we will have `onSave` simply add a new restaurant.

All we need to do is add a local restaurant `r` variable, populate it, and add it to the list:

```
private View.OnClickListener onSave=new View.OnClickListener() {
    public void onClick(View v) {
        Restaurant r=new Restaurant();
        EditText name=(EditText)findViewById(R.id.name);
        EditText address=(EditText)findViewById(R.id.addr);

        r.setName(name.getText().toString());
        r.setAddress(address.getText().toString());

        RadioGroup types=(RadioGroup)findViewById(R.id.types);

        switch (types.getCheckedRadioButtonId()) {
            case R.id.sit_down:
                r.setType("sit_down");
                break;

            case R.id.take_out:
                r.setType("take_out");
                break;

            case R.id.delivery:
                r.setType("delivery");
                break;
        }
    }
};
```

At this point, you should be able to rebuild and reinstall the application. Test it out to make sure that clicking the button does not cause any unexpected errors.

You will note that we are not adding the actual restaurant to anything – `r` is a local variable and so goes out of scope after `onClick()` returns. We will address this shortcoming later in this exercise.

Step #3: Implement `toString()`

To simplify the creation of our `ListView`, we need to have our restaurant class respond intelligently to `toString()`. That will be called on each restaurant as it is displayed in our list.

For the purposes of this tutorial, we will simply use the name – later tutorials will make the rows much more interesting and complex.

So, add a `toString()` implementation on `Restaurant` like this:

```
public String toString() {  
    return(getName());  
}
```

Recompile and ensure your application still builds.

Step #4: Add a `ListView` Widget

Now comes the challenging part – adding the `ListView` to the layout.

The challenge is in getting the layout right. Right now, while we have only the one screen to work with, we need to somehow squeeze in the list without eliminating space for anything else. In fact, ideally, the list takes up all the available space that is not being used by our current details form.

One way to achieve that is to use a `RelativeLayout` as the over-arching layout for the screen. We anchor the details form to the bottom of the screen, then have the list span the space from the top of the screen to the top of the details form.

To make this change, replace your current `LunchList/res/layout/main.xml` with the following:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <TableLayout android:id="@+id/details"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:stretchColumns="1"
        >
        <TableRow>
            <TextView android:text="Name:" />
            <EditText android:id="@+id/name" />
        </TableRow>
        <TableRow>
            <TextView android:text="Address:" />
            <EditText android:id="@+id/addr" />
        </TableRow>
        <TableRow>
            <TextView android:text="Type:" />
            <RadioGroup android:id="@+id/types">
                <RadioButton android:id="@+id/take_out"
                    android:text="Take-Out"
                />
                <RadioButton android:id="@+id/sit_down"
                    android:text="Sit-Down"
                />
                <RadioButton android:id="@+id/delivery"
                    android:text="Delivery"
                />
            </RadioGroup>
        </TableRow>
        <Button android:id="@+id/save"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="Save"
        />
    </TableLayout>
    <ListView android:id="@+id/restaurants"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_above="@+id/details"
    />
</RelativeLayout>
```

If you recompile and rebuild the application, then run it, you will see our form slid to the bottom, with empty space at the top:

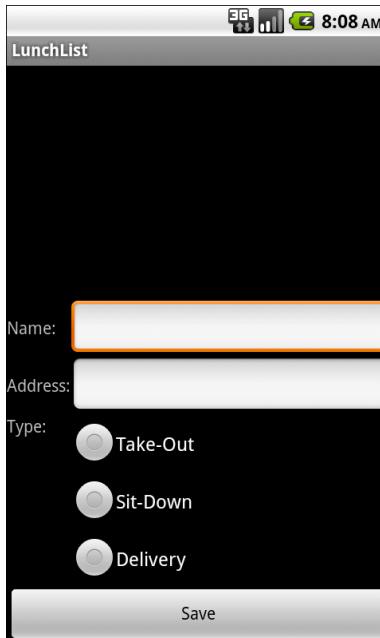


Figure 10. Adding a list to the top and sliding the form to the bottom

Step #5: Build and Attach the Adapter

The `ListView` will remain empty, of course, until we do something to populate it. What we want is for the list to show our running lineup of restaurant objects.

Since we have our `ArrayList<Restaurant>`, we can easily wrap it in an `ArrayAdapter<Restaurant>`. This also means, though, that when we add a restaurant, we need to add it to the `ArrayAdapter` via `add()` – the adapter will, in turn, put it in the `ArrayList`. Otherwise, if we add it straight to the `ArrayList`, the adapter will not know about the added restaurant and therefore will not display it.

Here is the new implementation of the `LunchList` class:

```
package apt.tutorial;  
  
import android.app.Activity;
```

```
import android.os.Bundle;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ListView;
import android.widget.RadioGroup;
import java.util.ArrayList;
import java.util.List;

public class LunchList extends Activity {
    List<Restaurant> model=new ArrayList<Restaurant>();
    ArrayAdapter<Restaurant> adapter=null;

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

        Button save=(Button)findViewById(R.id.save);
        save.setOnClickListener(onSave);

        ListView list=(ListView)findViewById(R.id.restaurants);

        adapter=new ArrayAdapter<Restaurant>(this,
            android.R.layout.simple_list_item_1,
            model);
        list.setAdapter(adapter);
    }

    private View.OnClickListener onSave=new View.OnClickListener() {
        public void onClick(View v) {
            Restaurant r=new Restaurant();
            EditText name=(EditText)findViewById(R.id.name);
            EditText address=(EditText)findViewById(R.id.addr);

            r.setName(name.getText().toString());
            r.setAddress(address.getText().toString());

            RadioGroup types=(RadioGroup)findViewById(R.id.types);

            switch (types.getCheckedRadioButtonId()) {
                case R.id.sit_down:
                    r.setType("sit_down");
                    break;

                case R.id.take_out:
                    r.setType("take_out");
                    break;

                case R.id.delivery:
                    r.setType("delivery");
                    break;
            }
        }
    };
}
```

```
        }  
  
        adapter.add(r);  
    };  
}
```

The magic value `android.R.layout.simple_list_item_1` is a stock layout for a list row, just displaying the text of the object in white on a black background with a reasonably large font. In later tutorials, we will change the look of our rows to suit our own designs.

If you then add a few restaurants via the form, it will look something like this:

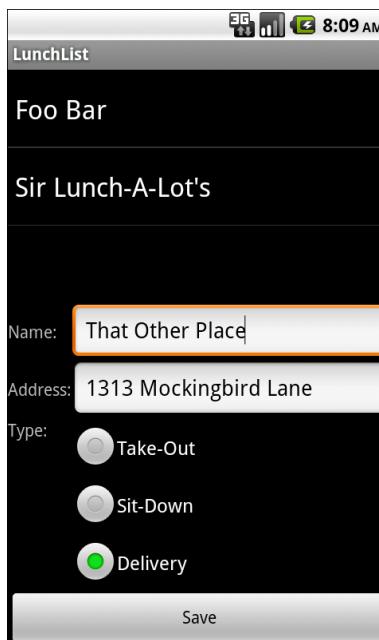


Figure 11. Our LunchList with a few fake restaurants added

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- See what the activity looks like if you use a `Spinner` instead of a `ListView`.
- Make the address field, presently an `EditText` widget, into an `AutoCompleteTextView`, using the other addresses as values to possibly reuse (e.g., for multiple restaurants in one place, such as a food court or mall).

Further Reading

Information on `ListView` and other selection widgets can be found in the "Using Selection Widgets" chapter of [The Busy Coder's Guide to Android Development](#).

TUTORIAL 5

Making Our List Be Fancy

In this tutorial, we will update the layout of our `ListView` rows, so they show both the name and address of the restaurant, plus an icon indicating the type. Along the way, we will need to create our own custom `ListAdapter` to handle our row views and a `RestaurantHolder` to populate a row from a restaurant.

Regarding the notion of adapters and `ListAdapter`, to quote from *The Busy Coder's Guide to Android Development*:

In the abstract, adapters provide a common interface to multiple disparate APIs. More specifically, in Android's case, adapters provide a common interface to the data model behind a selection-style widget, such as a listbox...Android's adapters are responsible for providing the roster of data for a selection widget plus converting individual elements of data into specific views to be displayed inside the selection widget.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the `04-ListView` edition of `LunchList` to use as a starting point.

Step #1: Create a Stub Custom Adapter

First, let us create a stub implementation of a `RestaurantAdapter` that will be where we put our logic for creating our own custom rows. That can look like this, implemented as an inner class of `LunchList`:

```
class RestaurantAdapter extends ArrayAdapter<Restaurant> {  
    RestaurantAdapter() {  
        super(LunchList.this,  
              android.R.layout.simple_list_item_1,  
              model);  
    }  
}
```

We hard-wire in the `android.R.layout.simple_list_item_1` layout for now, and we get our `Activity` and model from `LunchList` itself.

We also need to change our `adapter` data member to be a `RestaurantAdapter`, both where it is declared and where it is instantiated in `onCreate()`. Make these changes, then rebuild and reinstall the application and confirm it works as it did at the end of the previous tutorial.

Step #2: Design Our Row

Next, we want to design a row that incorporates all three of our model elements: name, address, and type. For the type, we will use three icons, one for each specific type (sit down, take-out, delivery). You can use whatever icons you wish, or you can get the icons used in this tutorial from the tutorial ZIP file that you can [download](#). They need to be named `ball_red.png`, `ball_yellow.png`, and `ball_green.png`, all located in `res/drawable/` in your project.

NOTE: If your project has no `res/drawable/` directory, but does have `res/drawable-ldpi/` and others with similar suffixes, rename `res/drawable-mdpi/` to `res/drawable/` directory for use in this project, and delete the other `res/drawable-*` directories.

The general layout is to have the icon on the left and the name stacked atop the address to the right:

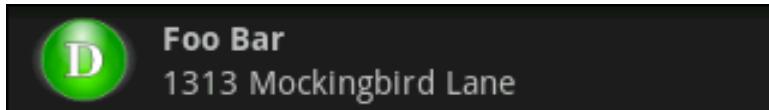


Figure 12. A fancy row for our fancy list

To achieve this look, we use a nested pair of `LinearLayout` containers. Use the following XML as the basis for `LunchList/res/layout/row.xml`:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:orientation="horizontal"  
    android:padding="4px"  
    >  
    <ImageView android:id="@+id/icon"  
        android:layout_width="wrap_content"  
        android:layout_height="fill_parent"  
        android:layout_alignParentTop="true"  
        android:layout_alignParentBottom="true"  
        android:layout_marginRight="4px"  
    />  
    <LinearLayout  
        android:layout_width="fill_parent"  
        android:layout_height="wrap_content"  
        android:orientation="vertical"  
        >  
        <TextView android:id="@+id/title"  
            android:layout_width="fill_parent"  
            android:layout_height="wrap_content"  
            android:layout_weight="1"  
            android:gravity="center_vertical"  
            android:textStyle="bold"  
            android:singleLine="true"  
            android:ellipsize="end"  
        />  
        <TextView android:id="@+id/address"  
            android:layout_width="fill_parent"  
            android:layout_height="wrap_content"  
            android:layout_weight="1"  
            android:gravity="center_vertical"  
            android:singleLine="true"  
            android:ellipsize="end"  
        />  
    </LinearLayout>  
</LinearLayout>
```

Some of the unusual attributes applied in this layout include:

- `android:padding`, which arranges for some whitespace to be put outside the actual widget contents but still be considered part of the widget (or container) itself when calculating its size
- `android:textStyle`, where we can indicate that some text is in bold or italics
- `android:singleLine`, which, if `true`, indicates that text should not word-wrap if it extends past one line
- `android:ellipsize`, which indicates where text should be truncated and ellipsized if it is too long for the available space

Step #3: Override `getView()`: The Simple Way

Next, we need to use this layout ourselves in our `RestaurantAdapter`. To do this, we need to override `getView()` and inflate the layout as needed for rows.

Modify `RestaurantAdapter` to look like the following:

```
class RestaurantAdapter extends ArrayAdapter<Restaurant> {
    RestaurantAdapter() {
        super(LunchList.this,
              android.R.layout.simple_list_item_1,
              model);
    }

    public View getView(int position, View convertView,
                        ViewGroup parent) {
        View row=convertView;

        if (row==null) {
            LayoutInflator inflater=getLayoutInflator();
            row=inflater.inflate(R.layout.row, null);
        }

        Restaurant r=model.get(position);

        ((TextView)row.findViewById(R.id.title)).setText(r.getName());
        ((TextView)row.findViewById(R.id.address)).setText(r.getAddress());

        ImageView icon=(ImageView)row.findViewById(R.id.icon);
```

```
if (r.getType().equals("sit_down")) {
    icon.setImageResource(R.drawable.ball_red);
}
else if (r.getType().equals("take_out")) {
    icon.setImageResource(R.drawable.ball_yellow);
}
else {
    icon.setImageResource(R.drawable.ball_green);
}

return(row);
}
```

Notice how we create a row only if needed, recycling existing rows. But, we still pick out each `TextView` and `ImageView` from each row and populate it from the restaurant at the indicated position.

Step #4: Create a RestaurantHolder

To improve performance and encapsulation, we should move the logic that populates a row from a restaurant into a separate class, one that can cache the `TextView` and `ImageView` widgets.

To do this, add the following static inner class to `LunchList`:

```
static class RestaurantHolder {
    private TextView name=null;
    private TextView address=null;
    private ImageView icon=null;
    private View row=null;

    RestaurantHolder(View row) {
        this.row=row;

        name=(TextView)row.findViewById(R.id.title);
        address=(TextView)row.findViewById(R.id.address);
        icon=(ImageView)row.findViewById(R.id.icon);
    }

    void populateFrom(Restaurant r) {
        name.setText(r.getName());
        address.setText(r.getAddress());

        if (r.getType().equals("sit_down")) {
            icon.setImageResource(R.drawable.ball_red);
        }
    }
}
```

```
    }
    else if (r.getType().equals("take_out")) {
        icon.setImageResource(R.drawable.ball_yellow);
    }
    else {
        icon.setImageResource(R.drawable.ball_green);
    }
}
```

Step #5: Recycle Rows via RestaurantHolder

To take advantage of the new `RestaurantHolder`, we need to modify `getView()` in `RestaurantAdapter`. Following the holder pattern, we need to create a `RestaurantHolder` when we inflate a new row, cache that wrapper in the row via `setTag()`, then get it back later via `getTag()`.

Change `getView()` to look like the following:

```
class RestaurantAdapter extends ArrayAdapter<Restaurant> {
    RestaurantAdapter() {
        super(LunchList.this, R.layout.row, model);
    }

    public View getView(int position, View convertView,
                        ViewGroup parent) {
        View row=convertView;
        RestaurantHolder holder=null;

        if (row==null) {
            LayoutInflater inflater=getLayoutInflater();

            row=inflater.inflate(R.layout.row, parent, false);
            holder=new RestaurantHolder(row);
            row.setTag(holder);
        }
        else {
            holder=(RestaurantHolder)row.getTag();
        }

        holder.populateFrom(model.get(position));

        return(row);
    }
}
```

This means the whole `LunchList` class looks like:

```
package apt.tutorial;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.view.LayoutInflater;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ImageView;
import android.widget.ListView;
import android.widget.RadioGroup;
import android.widget.TextView;
import java.util.ArrayList;
import java.util.List;

public class LunchList extends Activity {
    List<Restaurant> model=new ArrayList<Restaurant>();
    RestaurantAdapter adapter=null;

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

        Button save=(Button)findViewById(R.id.save);

        save.setOnClickListener(onSave);

        ListView list=(ListView)findViewById(R.id.restaurants);

        adapter=new RestaurantAdapter();
        list.setAdapter(adapter);
    }

    private View.OnClickListener onSave=new View.OnClickListener() {
        public void onClick(View v) {
            Restaurant r=new Restaurant();
            EditText name=(EditText)findViewById(R.id.name);
            EditText address=(EditText)findViewById(R.id.addr);

            r.setName(name.getText().toString());
            r.setAddress(address.getText().toString());

            RadioGroup types=(RadioGroup)findViewById(R.id.types);

            switch (types.getCheckedRadioButtonId()) {
                case R.id.sit_down:
                    r.setType("sit_down");
                    break;

                case R.id.take_out:
                    r.setType("take_out");
            }
        }
    };
}
```

```
        break;

    case R.id.delivery:
        r.setType("delivery");
        break;
    }

    adapter.add(r);
};

class RestaurantAdapter extends ArrayAdapter<Restaurant> {
    RestaurantAdapter() {
        super(LunchList.this, R.layout.row, model);
    }

    public View getView(int position, View convertView,
                        ViewGroup parent) {
        View row=convertView;
        RestaurantHolder holder=null;

        if (row==null) {
            LayoutInflator inflater=getLayoutInflater();

            row=inflater.inflate(R.layout.row, parent, false);
            holder=new RestaurantHolder(row);
            row.setTag(holder);
        }
        else {
            holder=(RestaurantHolder)row.getTag();
        }

        holder.populateFrom(model.get(position));

        return(row);
    }

    static class RestaurantHolder {
        private TextView name=null;
        private TextView address=null;
        private ImageView icon=null;
        private View row=null;

        RestaurantHolder(View row) {
            this.row=row;

            name=(TextView)row.findViewById(R.id.title);
            address=(TextView)row.findViewById(R.id.address);
            icon=(ImageView)row.findViewById(R.id.icon);
        }

        void populateFrom(Restaurant r) {
            name.setText(r.getName());
        }
    }
}
```

```
address.setText(r.getAddress());  
  
    if (r.getType().equals("sit_down")) {  
        icon.setImageResource(R.drawable.ball_red);  
    }  
    else if (r.getType().equals("take_out")) {  
        icon.setImageResource(R.drawable.ball_yellow);  
    }  
    else {  
        icon.setImageResource(R.drawable.ball_green);  
    }  
}  
}
```

Rebuild and reinstall the application, then try adding several restaurants and confirm that, when the list is scrolled, everything appears as it should – the name, address, and icon all change.

Note that you may experience a problem, where your `EditText` widgets shrink, failing to follow the `android:stretchColumns` rule. This is a [bug in Android](#) that will hopefully be repaired one day.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Customize the rows beyond just the icon based on each restaurant, such as applying different colors to the name based upon certain criteria.
- Use three different layouts for the three different restaurant types. To do this, you will need to override `getItemViewType()` and `getViewTypeCount()` in the custom adapter to return the appropriate data.

Further Reading

Using custom Adapter classes and creating list rows that are more than mere strings is covered in the "Getting Fancy with Lists" chapter of [The Busy Coder's Guide to Android Development](#).

Splitting the Tab

In this tutorial, we will move our `ListView` onto one tab and our form onto a separate tab of a `TabView`. Along the way, we will also arrange to update our form based on a `ListView` selections or clicks, even though the Save button will still only add new restaurants to our list.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the `05-FancyList` edition of `LunchList` to use as a starting point.

Step #1: Rework the Layout

First, we need to change our layout around, to introduce the tabs and split our UI between a list tab and a details tab. This involves:

- Removing the `RelativeLayout` and the layout attributes leveraging it, as that was how we had the list and form on a single screen
- Add in a `TabHost`, `TabWidget`, and `FrameLayout`, the latter of which is parent to the list and details

To accomplish this, replace your current `LunchList/res/layout/main.xml` with the following:

```
<?xml version="1.0" encoding="utf-8"?>
<TabHost xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/tabhost"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
        <TabWidget android:id="@+id/tabs"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
        />
        <FrameLayout android:id="@+id/tabcontent"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent"
        >
            <ListView android:id="@+id/restaurants"
                android:layout_width="fill_parent"
                android:layout_height="fill_parent"
            />
            <TableLayout android:id="@+id/details"
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:stretchColumns="1"
                android:paddingTop="4px"
            >
                <TableRow>
                    <TextView android:text="Name:" />
                    <EditText android:id="@+id/name" />
                </TableRow>
                <TableRow>
                    <TextView android:text="Address:" />
                    <EditText android:id="@+id/addr" />
                </TableRow>
                <TableRow>
                    <TextView android:text="Type:" />
                    <RadioGroup android:id="@+id/types">
                        <RadioButton android:id="@+id/take_out"
                            android:text="Take-Out"
                        />
                        <RadioButton android:id="@+id/sit_down"
                            android:text="Sit-Down"
                        />
                        <RadioButton android:id="@+id/delivery"
                            android:text="Delivery"
                        />
                    </RadioGroup>
                </TableRow>
                <Button android:id="@+id/save"
                    android:layout_width="fill_parent"
                    android:layout_height="wrap_content"
                    android:text="Save"
                />
            
```

```
</TableLayout>
</FrameLayout>
</LinearLayout>
</TabHost>
```

Step #2: Wire In the Tabs

Next, we need to modify the `LunchList` itself, so it is a `TabActivity` (rather than a plain `Activity`) and teaches the `TabHost` how to use our `FrameLayout` contents for the individual tab panes. To do this:

1. Add imports to `LunchList` for `android.app.TabActivity` and `android.widget.TabHost`
2. Make `LunchList` extend `TabActivity`
3. Obtain 32px high icons from some source to use for the list and details tab icons, place them in `LunchList/res/drawable` as `list.png` and `restaurant.png`, respectively
4. Add the following code to the end of your `onCreate()` method:

```
TabHost.TabSpec spec=getTabHost().newTabSpec("tag1");
spec.setContent(R.id.restaurants);
spec.setIndicator("List", getResources()
    .getDrawable(R.drawable.list));
getTabHost().addTab(spec);

spec=getTabHost().newTabSpec("tag2");
spec.setContent(R.id.details);
spec.setIndicator("Details", getResources()
    .getDrawable(R.drawable.restaurant));
getTabHost().addTab(spec);

getTabHost().setCurrentTab(0);
```

At this point, you can recompile and reinstall the application and try it out. You should see a two-tab UI like this:

Splitting the Tab

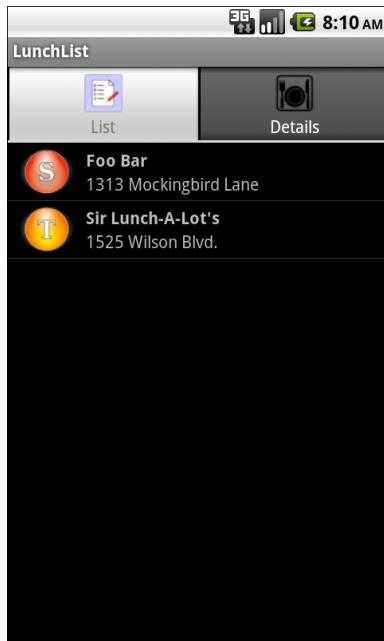


Figure 13. The first tab of the two-tab LunchList

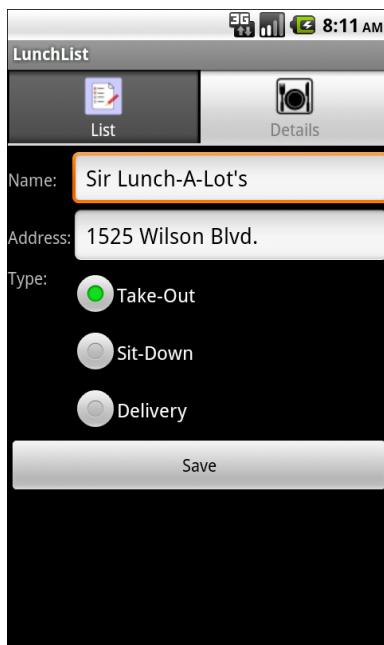


Figure 14. The second tab of the two-tab LunchList

Step #3: Get Control On List Events

Next, we need to detect when the user clicks on one of our restaurants in the list, so we can update our detail form with that information.

First, add an import for `android.widget.AdapterView` to `LunchList`. Then, create an `AdapterView.OnItemClickListener` named `onListClick`:

```
private AdapterView.OnItemClickListener onListClick=new
AdapterView.OnItemClickListener() {
    public void onItemClick(AdapterView<?> parent,
                        View view, int position,
                        long id) {
}
};
```

Finally, call `setOnItemClickListener()` on the `ListView` in the activity's `onCreate()` to connect the `ListView` to the `onListClick` listener object (`list.setOnItemClickListener(onListClick);`)

Step #4: Update Our Restaurant Form On Clicks

Next, now that we have control in a list item click, we need to actually find the associated restaurant and update our details form.

To do this, you need to do two things. First, move the `name`, `address`, and `types` variables into data members and populate them in the activity's `onCreate()` – our current code has them as local variables in the `onSave` listener object's `onClick()` method. So, you should have some data members like:

```
EditText name=null;
EditText address=null;
RadioGroup types=null;
```

And some code after the call to `setContentView()` in `onCreate()` like:

```
name=(EditText)findViewById(R.id.name);
address=(EditText)findViewById(R.id.addr);
types=(RadioGroup)findViewById(R.id.types);
```

Then, add smarts to `onListClick` to update the details form:

```
private AdapterView.OnItemClickListener onListClick=new
AdapterView.OnItemClickListener() {
    public void onItemClick(AdapterView<?> parent,
                        View view, int position,
                        long id) {
        Restaurant r=model.get(position);

        name.setText(r.getName());
        address.setText(r.getAddress());

        if (r.getType().equals("sit_down")) {
            types.check(R.id.sit_down);
        }
        else if (r.getType().equals("take_out")) {
            types.check(R.id.take_out);
        }
        else {
            types.check(R.id.delivery);
        }
    }
};
```

Note how we find the clicked-upon restaurant via the `position` parameter, which is an index into our `ArrayList` of restaurants.

Step #5: Switch Tabs On Clicks

Finally, we want to switch to the details form when the user clicks a restaurant in the list.

This is just one extra line of code, in the `onItemClick()` method of our `onListClick` listener object:

```
getTabHost().setCurrentTab(1);
```

This just changes the current tab to the one known as index 1, which is the second tab (tabs start counting at 0).

At this point, you should be able to recompile and reinstall the application and test out the new functionality.

Here is the complete source code to our `LunchList` activity, after all of the changes made in this tutorial:

```
package apt.tutorial;

import android.app.TabActivity;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.view.LayoutInflater;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ImageView;
import android.widget.ListView;
import android.widget.RadioGroup;
import android.widget.TabHost;
import android.widget.TextView;
import java.util.ArrayList;
import java.util.List;
```

```
public class LunchList extends TabActivity {
    List<Restaurant> model=new ArrayList<Restaurant>();
    RestaurantAdapter adapter=null;
    EditText name=null;
    EditText address=null;
    RadioGroup types=null;

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

        name=(EditText)findViewById(R.id.name);
        address=(EditText)findViewById(R.id.addr);
        types=(RadioGroup)findViewById(R.id.types);

        Button save=(Button)findViewById(R.id.save);
        save.setOnClickListener(onSave);

        ListView list=(ListView)findViewById(R.id.restaurants);

        adapter=new RestaurantAdapter();
        list.setAdapter(adapter);

        TabHost.TabSpec spec=getTabHost().newTabSpec("tag1");

        spec.setContent(R.id.restaurants);
        spec.setIndicator("List", getResources()
            .getDrawable(R.drawable.list));
        getTabHost().addTab(spec);
```

```
spec=getTabHost().newTabSpec("tag2");
spec.setContent(R.id.details);
spec.setIndicator("Details", getResources()
    .getDrawable(R.drawable.restaurant));
getTabHost().addTab(spec);

getTabHost().setCurrentTab(0);

list.setOnItemClickListener(onListClick);
}

private View.OnClickListener onSave=new View.OnClickListener() {
    public void onClick(View v) {
        Restaurant r=new Restaurant();
        r.setName(name.getText().toString());
        r.setAddress(address.getText().toString());

        switch (types.getCheckedRadioButtonId()) {
            case R.id.sit_down:
                r.setType("sit_down");
                break;

            case R.id.take_out:
                r.setType("take_out");
                break;

            case R.id.delivery:
                r.setType("delivery");
                break;
        }

        adapter.add(r);
    }
};

private AdapterView.OnItemClickListener onListClick=new
AdapterView.OnItemClickListener() {
    public void onItemClick(AdapterView<?> parent,
        View view, int position,
        long id) {
        Restaurant r=model.get(position);

        name.setText(r.getName());
        address.setText(r.getAddress());

        if (r.getType().equals("sit_down")) {
            types.check(R.id.sit_down);
        }
        else if (r.getType().equals("take_out")) {
            types.check(R.id.take_out);
        }
        else {
            types.check(R.id.delivery);
        }
    }
};
```

```
    }

    getTabHost().setCurrentTab(1);
}

};

class RestaurantAdapter extends ArrayAdapter<Restaurant> {
    RestaurantAdapter() {
        super(LunchList.this, R.layout.row, model);
    }

    public View getView(int position, View convertView,
                        ViewGroup parent) {
        View row=convertView;
        RestaurantHolder holder=null;

        if (row==null) {
            LayoutInflater inflater=getLayoutInflater();

            row=inflater.inflate(R.layout.row, parent, false);
            holder=new RestaurantHolder(row);
            row.setTag(holder);
        }
        else {
            holder=(RestaurantHolder)row.getTag();
        }

        holder.populateFrom(model.get(position));

        return(row);
    }
}

static class RestaurantHolder {
    private TextView name=null;
    private TextView address=null;
    private ImageView icon=null;
    private View row=null;

    RestaurantHolder(View row) {
        this.row=row;

        name=(TextView)row.findViewById(R.id.title);
        address=(TextView)row.findViewById(R.id.address);
        icon=(ImageView)row.findViewById(R.id.icon);
    }

    void populateFrom(Restaurant r) {
        name.setText(r.getName());
        address.setText(r.getAddress());

        if (r.getType().equals("sit_down")) {
            icon.setImageResource(R.drawable.ball_red);
        }
    }
}
```

```
        else if (r.getType().equals("take_out")) {
            icon.setImageResource(R.drawable.ball_yellow);
        }
        else {
            icon.setImageResource(R.drawable.ball_green);
        }
    }
}
```

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Add a date in the restaurant model to note the last time you visited the restaurant, then use either `DatePicker` or `DatePickerDialog` to allow users to set the date when they create their restaurant objects.
- Try making a version of the activity that uses a `ViewFlipper` and a `Button` to flip from the list to the detail form, rather than using two tabs.

Further Reading

The use of tabs in an Android activity is covered in the "Employing Fancy Widgets" chapter of [The Busy Coder's Guide to Android Development](#).

Menus and Messages

In this tutorial, we will add a `EditText` for a note to our details form and restaurant model. Then, we will add an options menu that will display the note as a `Toast`.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 06-Tabs edition of `LunchList` to use as a starting point.

Step #1: Add Notes to the Restaurant

First, our restaurant model does not have any spot for notes. Add a `String` notes data member plus an associated getter and setter. Your resulting class should look like:

```
package apt.tutorial;

public class Restaurant {
    private String name="";
    private String address="";
    private String type="";
    private String notes="";

    public String getName() {
        return(name);
    }
}
```

```
}

public void setName(String name) {
    this.name=name;
}

public String getAddress() {
    return(address);
}

public void setAddress(String address) {
    this.address=address;
}

public String getType() {
    return(type);
}

public void setType(String type) {
    this.type=type;
}

public String getNotes() {
    return(notes);
}

public void setNotes(String notes) {
    this.notes=notes;
}

public String toString() {
    return(getName());
}
}
```

Step #2: Add Notes to the Detail Form

Next, we need `LunchList` to make use of the notes. To do this, first add the following `TableRow` above the `Save` button in our `TableLayout` in `LunchList/res/layout/main.xml`:

```
<TableRow>
    <TextView android:text="Notes:" />
    <EditText android:id="@+id/notes"
        android:singleLine="false"
        android:gravity="top"
        android:lines="2"
        android:scrollHorizontally="false"
        android:maxLines="2"
        android:maxWidth="200sp"
```

```
/>  
</TableRow>
```

Then, we need to modify the `LunchList` activity itself, by:

1. Adding another data member for the notes `EditText` widget defined above
2. Find our notes `EditText` widget as part of `onCreate()`, like we do with other `EditText` widgets
3. Save our notes to our restaurant in `onSave`
4. Restore our notes to the `EditText` in `onListClick`

At this point, you can recompile and reinstall the application to see your notes field in action:

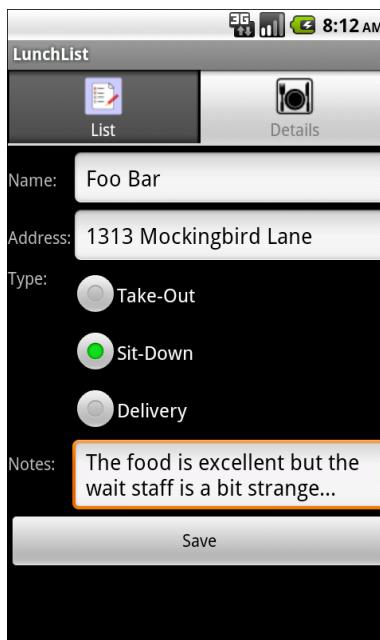


Figure 15. The notes field in the details form

Step #3: Define the Option Menu

Now, we need to create an option menu and arrange for it to be displayed when the user clicks the [MENU] button.

The menu itself can be defined as a small piece of XML. Enter the following as `LunchList/res/menu/option.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/toast"
        android:title="Raise Toast"
        android:icon="@drawable/toast"
    />
</menu>
```

This code relies upon an icon stored in `LunchList/res/drawable/toast.png`. Find something suitable to use, preferably around 32px high.

Then, to arrange for the menu to be displayed, add the following method to `LunchList`:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    new MenuInflater(this).inflate(R.menu.option, menu);

    return(super.onCreateOptionsMenu(menu));
}
```

Note that you will also need to define imports for `android.view.Menu` and `android.view.MenuInflater` for this to compile cleanly.

At this point, you can rebuild and reinstall the application. Click the [MENU] button, from either tab, to see the option menu with its icon:

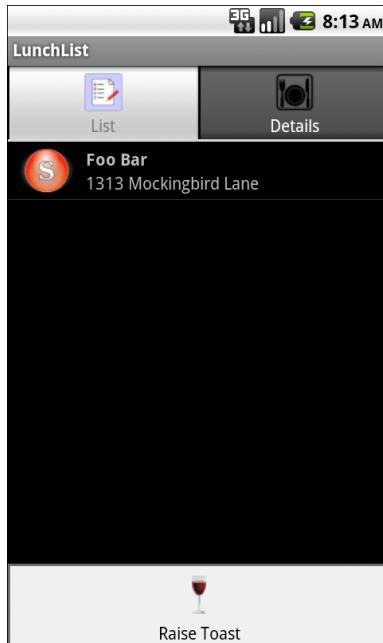


Figure 16. The LunchList option menu, displayed, with one menu choice

Step #4: Show the Notes as a Toast

Finally, we need to get control when the user selects the Raise Toast menu choice and display the notes in a `Toast`.

The problem is that, to do this, we need to know what restaurant to show. So far, we have not been holding onto a specific restaurant except when we needed it, such as when we populate the details form. Now, we need to know our current restaurant, defined as the one visible in the detail form...which could be none, if we have not yet saved anything in the form.

To make all of this work, do the following:

1. Add another data member, `restaurant current`, to hold the current restaurant. Be sure to initialize it to `null`.
2. In `onSave` and `onListClick`, rather than declaring local `restaurant` variables, use `current` to hold the restaurant we are saving (in

`onSave`) or have clicked on (in `onListClick`). You will need to change all references to the old `r` variable to be `current` in these two objects.

3. Add imports for `android.view.MenuItem` and `android.widget.Toast`.
4. Add the following implementation of `onOptionsItemSelected()` to your `LunchList` class:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.toast) {
        String message="No restaurant selected";

        if (current!=null) {
            message=current.getNotes();
        }

        Toast.makeText(this, message, Toast.LENGTH_LONG).show();

        return(true);
    }

    return(super.onOptionsItemSelected(item));
}
```

Note how we will either display "No restaurant selected" (if `current` is `null`) or the restaurant's notes, depending on our current state.

You can now rebuild and reinstall the application. Enter and save a restaurant, with notes, then choose the Raise Toast option menu item, and you will briefly see your notes in a `Toast`:

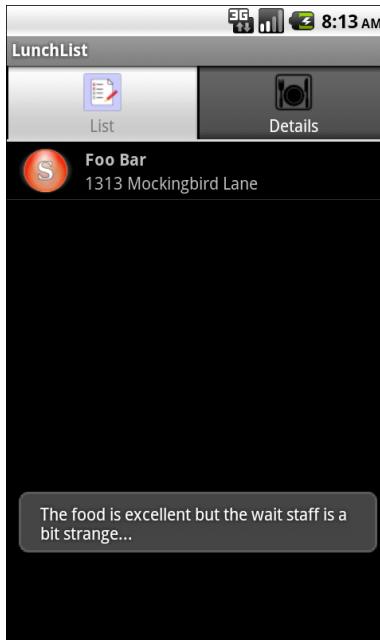


Figure 17. The Toast displayed, with some notes

The LunchList activity, as a whole, is shown below, incorporating all of the changes outlined in this tutorial:

```
package apt.tutorial;

import android.app.TabActivity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.view.ViewGroup;
import android.view.LayoutInflater;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ImageView;
import android.widget.ListView;
import android.widget.RadioGroup;
import android.widget.TabHost;
import android.widget.TextView;
import android.widget.Toast;
import java.util.ArrayList;
import java.util.List;
```

```
public class LunchList extends TabActivity {
    List<Restaurant> model=new ArrayList<Restaurant>();
    RestaurantAdapter adapter=null;
    EditText name=null;
    EditText address=null;
    EditText notes=null;
    RadioGroup types=null;
    Restaurant current=null;

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

        name=(EditText)findViewById(R.id.name);
        address=(EditText)findViewById(R.id.addr);
        notes=(EditText)findViewById(R.id.notes);
        types=(RadioGroup)findViewById(R.id.types);

        Button save=(Button)findViewById(R.id.save);
        save.setOnClickListener(onSave);

        ListView list=(ListView)findViewById(R.id.restaurants);

        adapter=new RestaurantAdapter();
        list.setAdapter(adapter);

        TabHost.TabSpec spec=getTabHost().newTabSpec("tag1");

        spec.setContent(R.id.restaurants);
        spec.setIndicator("List", getResources()
            .getDrawable(R.drawable.list));
        getTabHost().addTab(spec);

        spec=getTabHost().newTabSpec("tag2");
        spec.setContent(R.id.details);
        spec.setIndicator("Details", getResources()
            .getDrawable(R.drawable.restaurant));
        getTabHost().addTab(spec);

        getTabHost().setCurrentTab(0);

        list.setOnItemClickListener(onListClick);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        new MenuInflater(this).inflate(R.menu.option, menu);

        return(super.onCreateOptionsMenu(menu));
    }
}
```

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.toast) {
        String message="No restaurant selected";

        if (current!=null) {
            message=current.getNotes();
        }

        Toast.makeText(this, message, Toast.LENGTH_LONG).show();

        return(true);
    }

    return(super.onOptionsItemSelected(item));
}

private View.OnClickListener onSave=new View.OnClickListener() {
    public void onClick(View v) {
        current=new Restaurant();
        current.setName(name.getText().toString());
        current.setAddress(address.getText().toString());
        current.setNotes(notes.getText().toString());

        switch (types.getCheckedRadioButtonId()) {
            case R.id.sit_down:
                current.setType("sit_down");
                break;

            case R.id.take_out:
                current.setType("take_out");
                break;

            case R.id.delivery:
                current.setType("delivery");
                break;
        }

        adapter.add(current);
    }
};

private AdapterView.OnItemClickListener onListClick=new
AdapterView.OnItemClickListener() {
    public void onItemClick(AdapterView<?> parent,
                        View view, int position,
                        long id) {
        current=model.get(position);

        name.setText(current.getName());
        address.setText(current.getAddress());
        notes.setText(current.getNotes());
    }
};
```

```
        if (current.getType().equals("sit_down")) {
            types.check(R.id.sit_down);
        }
        else if (current.getType().equals("take_out")) {
            types.check(R.id.take_out);
        }
        else {
            types.check(R.id.delivery);
        }

        getTabHost().setCurrentTab(1);
    }
};

class RestaurantAdapter extends ArrayAdapter<Restaurant> {
    RestaurantAdapter() {
        super(LunchList.this, R.layout.row, model);
    }

    public View getView(int position, View convertView,
                        ViewGroup parent) {
        View row=convertView;
        RestaurantHolder holder=null;

        if (row==null) {
            LayoutInflater inflater=getLayoutInflater();

            row=inflater.inflate(R.layout.row, parent, false);
            holder=new RestaurantHolder(row);
            row.setTag(holder);
        }
        else {
            holder=(RestaurantHolder)row.getTag();
        }

        holder.populateFrom(model.get(position));

        return(row);
    }
}

static class RestaurantHolder {
    private TextView name=null;
    private TextView address=null;
    private ImageView icon=null;
    private View row=null;

    RestaurantHolder(View row) {
        this.row=row;

        name=(TextView)row.findViewById(R.id.title);
        address=(TextView)row.findViewById(R.id.address);
        icon=(ImageView)row.findViewById(R.id.icon);
    }
}
```

```
void populateFrom(Restaurant r) {  
    name.setText(r.getName());  
    address.setText(r.getAddress());  
  
    if (r.getType().equals("sit_down")) {  
        icon.setImageResource(R.drawable.ball_red);  
    }  
    else if (r.getType().equals("take_out")) {  
        icon.setImageResource(R.drawable.ball_yellow);  
    }  
    else {  
        icon.setImageResource(R.drawable.ball_green);  
    }  
}
```

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Try using an `AlertDialog` instead of a `Toast` to display the message.
- Try adding a menu option to switch you between tabs. In particular, change the text and icon on the menu option to reflect the other tab (i.e., on the List tab, the menu should show "Details" and the details tab icon; on the Details tab, the menu should show "List" and the List tab icon).
- Try creating an `ErrorDialog` designed to display exceptions in a "pleasant" format to the end user. The `ErrorDialog` should also log the exceptions via `android.util.Log`. Use some sort of runtime exception (e.g., division by zero) for generating exceptions to pass to the dialog.

Further Reading

You can learn more about menus – both option menus and context menus – in the "Applying Menus" chapter of [The Busy Coder's Guide to Android Development](#). The use of a `Toast` is covered in the "Showing Pop-Up Messages" chapter of the same book.

Sitting in the Background

In this tutorial, we will simulate having the `LunchList` do some background processing in a secondary thread, updating the user interface via a progress bar. While all of these tutorials are somewhat contrived, this one will be more contrived than most, as there is not much we are really able to do in a `LunchList` that would even require long processing in a background thread. So, please forgive us if this tutorial is a bit goofy.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the `07-MenusMessages` edition of `LunchList` to use as a starting point.

Step #1: Initialize the Progress Bar

For this application, rather than use a `ProgressBar` widget, we will use the progress bar feature of the `Activity` window. This will put a progress bar in the title bar, rather than clutter up our layouts.

This requires a bit of initialization. Specifically, we need to add a line to `onCreate()` that will request this feature be activated. We have to do this before calling `setContentView()`, so we add it right after chaining to the superclass:

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

    name=(EditText)findViewById(R.id.name);
    address=(EditText)findViewById(R.id.addr);
    notes=(EditText)findViewById(R.id.notes);
    types=(RadioGroup)findViewById(R.id.types);

    Button save=(Button)findViewById(R.id.save);
    save.setOnClickListener(onSave);

    ListView list=(ListView)findViewById(R.id.restaurants);
    adapter=new RestaurantAdapter();
    list.setAdapter(adapter);

    TabHost.TabSpec spec=getTabHost().newTabSpec("tag1");
    spec.setContent(R.id.restaurants);
    spec.setIndicator("List", getResources()
        .getDrawable(R.drawable.list));
    getTabHost().addTab(spec);

    spec=getTabHost().newTabSpec("tag2");
    spec.setContent(R.id.details);
    spec.setIndicator("Details", getResources()
        .getDrawable(R.drawable.restaurant));
    getTabHost().addTab(spec);

    getTabHost().setCurrentTab(0);
    list.setOnItemClickListener(onListClick);
}
```

Also, add another data member, an `int` named `progress`.

Step #2: Create the Work Method

The theory of this demo is that we have something that takes a long time, and we want to have that work done in a background thread and update the progress along the way. So, the first step is to build something that will run a long time.

To do that, first, implement a `doSomeLongWork()` method on `LunchList` as follows:

```
private void doSomeLongWork(final int incr) {
    SystemClock.sleep(250); // should be something more useful!
}
```

Here, we sleep for 250 milliseconds, simulating doing some meaningful work.

Then, create a private `Runnable` in `LunchList` that will fire off `doSomeLongWork()` a number of times, as follows:

```
private Runnable longTask=new Runnable() {
    public void run() {
        for (int i=0;i<20;i++) {
            doSomeLongWork(500);
        }
    }
};
```

Here, we just loop 20 times, so the overall background thread will run for 5 seconds.

Step #3: Fork the Thread from the Menu

Next, we need to arrange to do this (fake) long work at some point. The easiest way to do that is add another menu choice. Update the `LunchList/res/menu/option.xml` file to look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/toast"
          android:title="Raise Toast"
          android:icon="@drawable/toast"
    />
    <item android:id="@+id/run"
          android:title="Run Long Task"
          android:icon="@drawable/run"
    />
</menu>
```

This requires a graphic image in `LunchList/res/drawable/run.png` – find something that you can use that is around 32px high.

Since the menu item is in the menu XML, we do not need to do anything special to display the item – it will just be added to the menu automatically. We do, however, need to arrange to do something useful when the menu choice is chosen. So, update `onOptionsItemSelected()` in `LunchList` to look like the following:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.toast) {
        String message="No restaurant selected";

        if (current!=null) {
            message=current.getNotes();
        }

        Toast.makeText(this, message, Toast.LENGTH_LONG).show();

        return(true);
    }
    else if (item.getItemId()==R.id.run) {
        new Thread(longTask).start();
    }

    return(super.onOptionsItemSelected(item));
}
```

You are welcome to recompile, reinstall, and run the application. However, since our background thread does not do anything visible at the moment, all you will see that is different is the new menu item:

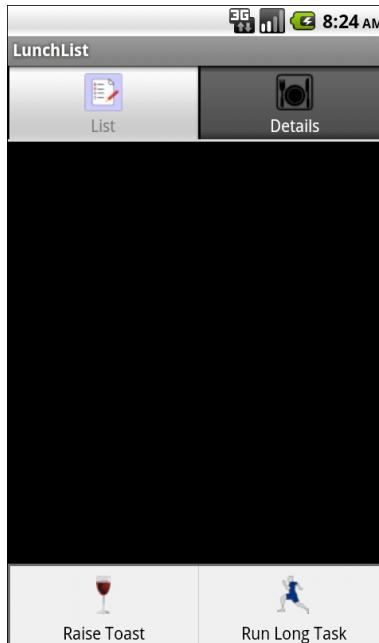


Figure 18. The Run Long Task menu item

Step #4: Manage the Progress Bar

Finally, we need to actually make use of the progress indicator. This involves making it visible when we start our long-running task, updating it as the task proceeds, and hiding it again when the task is complete.

First, make it visible by updating `onOptionsItemSelected()` to show it:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.toast) {
        String message="No restaurant selected";

        if (current!=null) {
            message=current.getNotes();
        }

        Toast.makeText(this, message, Toast.LENGTH_LONG).show();

        return(true);
    }
    else if (item.getItemId()==R.id.run) {
```

```
    setProgressBarVisibility(true);
    progress=0;
    new Thread(longTask).start();

    return(true);
}

return(super.onOptionsItemSelected(item));
}
```

Notice the extra line that makes progress visible.

Then, we need to update the progress bar on each pass, so make this change to doSomeLongWork():

```
private void doSomeLongWork(final int incr) {
    runOnUiThread(new Runnable() {
        public void run() {
            progress+=incr;
            setProgress(progress);
        }
    });

    SystemClock.sleep(250); // should be something more useful!
}
```

Notice how we use runOnUiThread() to make sure our progress bar update occurs on the UI thread.

Finally, we need to hide the progress bar when we are done, so make this change to our longTask Runnable:

```
private Runnable longTask=new Runnable() {
    public void run() {
        for (int i=0;i<20;i++) {
            doSomeLongWork(500);
        }

        runOnUiThread(new Runnable() {
            public void run() {
                setProgressBarVisibility(false);
            }
        });
    }
};
```

At this point, you can rebuild, reinstall, and run the application. When you choose the Run Long Task menu item, you will see the progress bar appear for five seconds, progressively updated as the "work" gets done:

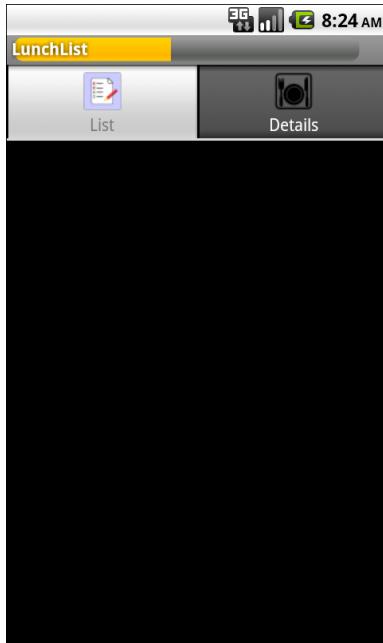


Figure 19. The progress bar in action

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Have the background thread also update some UI element when the work is completed, beyond dismissing the progress bar. Make sure you arrange to update the UI on the UI thread!
- Instead of using `Activity#runOnUiThread()`, try using a `Handler` for communication between the background thread and the UI thread.
- Instead of starting a `Thread` from the menu choice, have the `Thread` be created in `onCreate()` and have it monitor a `LinkedBlockingQueue` (from `java.util.concurrent`) as a source of work to be done. Create a `FakeJob` that does what our current long-running method does, and

a `KillJob` that causes the `Thread` to fall out of its queue-monitoring loop.

Further Reading

Coverage of the Android concept of "the UI thread" and tools like the Handler for managing communication between threads can be found in the "Dealing with Threads" chapter of [The Busy Coder's Guide to Android Development](#). You will also learn about `AsyncTask` in that chapter, which is another important means of coordinating background and UI thread operations.

If you are interested in Java threading in general, particularly the use of the `java.util.concurrent` set of thread-management classes, the book [Java Concurrency in Practice](#) is a popular source of information.

In this tutorial, we will make our background task take a bit longer, then arrange to pause the background work when we start up another activity and restart the background work when our activity regains control. This pattern – stopping unnecessary background work when the activity is paused – is a good design pattern and is not merely something used for a tutorial.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 08-Threads edition of `LunchList` to use as a starting point.

Step #1: Lengthen the Background Work

First, let us make the background work take a bit longer, so we have a bigger "window" in which to test whether our pause-and-resume logic works. It is also helpful, in our case, to synchronize our loop with our progress, so rather than counting `0` to `20` by `1`, we should count from `0` to `10000` by `200`, so the loop counter and progress are the same.

In the `longTask Runnable`, change the loop to look like this:

```
for (int i=progress;  
     i<10000;  
     i+=200) {  
    doSomeLongWork(200);  
}
```

Step #2: Pause in onPause()

Now, we need to arrange to have our thread stop running when the activity is paused (e.g., some other activity has taken over the screen). Since threads are relatively cheap to create and destroy, we can simply have our current running thread stop and start a fresh one, if needed, in onResume().

While there are some deprecated methods on Thread to try to forcibly terminate them, it is generally better to let the Thread stop itself by falling out of whatever processing loop it is in. So, what we want to do is let the background thread know the activity is not active.

To do this, first import java.util.concurrent.atomic.AtomicBoolean in LunchList and add an AtomicBoolean data member named isActive, initially set to true (new AtomicBoolean(true);).

Then, in the longTask Runnable, change the loop to also watch for the state of isActive, falling out of the loop if the activity is no longer active:

```
for (int i=progress;  
     i<10000 && isActive.get();  
     i+=200) {  
    doSomeLongWork(200);  
}
```

Finally, implement onPause() to update the state of isActive:

```
@Override  
public void onPause() {  
    super.onPause();  
  
    isActive.set(false);  
}
```

Note how we chain to the superclass in `onPause()` – if we fail to do this, we will get a runtime error.

With this implementation, our background thread will run to completion or until `isActive` is `false`, whichever comes first.

Step #3: Resume in `onResume()`

Now, we need to restart our thread if it is needed. It will be needed if the progress is greater than `0`, indicating we were in the middle of our background work when our activity was so rudely interrupted.

So, add the following implementation of `onResume()`:

```
@Override  
public void onResume() {  
    super.onResume();  
  
    isActive.set(true);  
  
    if (progress>0) {  
        startWork();  
    }  
}
```

This assumes we have pulled out our thread-starting logic into a `startWork()` method, which you should implement as follows:

```
private void startWork() {  
    setProgressBarVisibility(true);  
    new Thread(longTask).start();  
}
```

And you can change our menu handler to also use `startWork()`:

```
@Override  
public boolean onOptionsItemSelected(MenuItem item) {  
    if (item.getItemId()==R.id.toast) {  
        String message="No restaurant selected";  
  
        if (current!=null) {  
            message=current.getNotes();  
        }  
    }  
}
```

```
    Toast.makeText(this, message, Toast.LENGTH_LONG).show();

    return(true);
}
else if (item.getItemId()==R.id.run) {
    startWork();

    return(true);
}

return(super.onOptionsItemSelected(item));
}
```

Finally, we need to not reset and hide the progress indicator when our background thread ends if it ends because our activity is not active. Otherwise, we will never restart it, since the progress will be reset to 0 every time. So, change longTask one more time, to look like this:

```
private Runnable longTask=new Runnable() {
    public void run() {
        for (int i=progress;
             i<10000 && isActive.get();
             i+=200) {
            doSomeLongWork(200);
        }

        if (isActive.get()) {
            runOnUiThread(new Runnable() {
                public void run() {
                    setProgressBarVisibility(false);
                    progress=0;
                }
            });
        }
    }
};
```

What this does is reset the progress only if we are active when the work is complete, so we are ready for the next round of work. If we are inactive, and fell out of our loop for that reason, we leave the progress as-is.

At this point, recompile and reinstall the application. To test this feature:

1. Use the [MENU] button to run the long task.

2. While it is running, click the green phone button on the emulator (lower-left corner of the "phone"). This will bring up the call log activity and, as a result, pause our `LunchList` activity.
3. After a while, click the BACK button – you should see the `LunchList` resuming the background work from the point where it left off.

Here is the full `LunchList` implementation, including the changes shown above:

```
package apt.tutorial;

import android.app.TabActivity;
import android.os.Bundle;
import android.os.SystemClock;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.view.ViewGroup;
import android.view.LayoutInflater;
import android.view.Window;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ImageView;
import android.widget.ListView;
import android.widget.RadioGroup;
import android.widget.TabHost;
import android.widget.TextView;
import android.widget.Toast;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.AtomicBoolean;

public class LunchList extends TabActivity {
    List<Restaurant> model=new ArrayList<Restaurant>();
    RestaurantAdapter adapter=null;
    EditText name=null;
    EditText address=null;
    EditText notes=null;
    RadioGroup types=null;
    Restaurant current=null;
    AtomicBoolean isActive=new AtomicBoolean(true);
    int progress=0;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_PROGRESS);
```

```
setContentView(R.layout.main);

name=(EditText)findViewById(R.id.name);
address=(EditText)findViewById(R.id.addr);
notes=(EditText)findViewById(R.id.notes);
types=(RadioGroup)findViewById(R.id.types);

Button save=(Button)findViewById(R.id.save);
save.setOnClickListener(onSave);

ListView list=(ListView)findViewById(R.id.restaurants);

adapter=new RestaurantAdapter();
list.setAdapter(adapter);

TabHost.TabSpec spec=getTabHost().newTabSpec("tag1");

spec.setContent(R.id.restaurants);
spec.setIndicator("List", getResources()
    .getDrawable(R.drawable.list));
getTabHost().addTab(spec);

spec=getTabHost().newTabSpec("tag2");
spec.setContent(R.id.details);
spec.setIndicator("Details", getResources()
    .getDrawable(R.drawable.restaurant));
getTabHost().addTab(spec);

getTabHost().setCurrentTab(0);

list.setOnItemClickListener(onListClick);
}

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

    isActive.set(false);
}

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

    isActive.set(true);

    if (progress>0) {
        startWork();
    }
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
```

```
new MenuInflater(this).inflate(R.menu.option, menu);

    return(super.onCreateOptionsMenu(menu));
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.toast) {
        String message="No restaurant selected";

        if (current!=null) {
            message=current.getNotes();
        }

        Toast.makeText(this, message, Toast.LENGTH_LONG).show();

        return(true);
    }
    else if (item.getItemId()==R.id.run) {
        startWork();

        return(true);
    }

    return(super.onOptionsItemSelected(item));
}

private void startWork() {
    setProgressBarVisibility(true);
    new Thread(longTask).start();
}

private void doSomeLongWork(final int incr) {
    runOnUiThread(new Runnable() {
        public void run() {
            progress+=incr;
            setProgress(progress);
        }
    });

    SystemClock.sleep(250); // should be something more useful!
}

private View.OnClickListener onSave=new View.OnClickListener() {
    public void onClick(View v) {
        current=new Restaurant();
        current.setName(name.getText().toString());
        current.setAddress(address.getText().toString());
        current.setNotes(notes.getText().toString());

        switch (types.getCheckedRadioButtonId()) {
            case R.id.sit_down:
                current.setType("sit_down");
            case R.id.take_out:
                current.setType("take_out");
        }
    }
}
```

```
        break;

    case R.id.take_out:
        current.setType("take_out");
        break;

    case R.id.delivery:
        current.setType("delivery");
        break;
    }

    adapter.add(current);
}
};

private AdapterView.OnItemClickListener onListClick=new
AdapterView.OnItemClickListener() {
    public void onItemClick(AdapterView<?> parent,
                           View view, int position,
                           long id) {
        current=model.get(position);

        name.setText(current.getName());
        address.setText(current.getAddress());
        notes.setText(current.getNotes());

        if (current.getType().equals("sit_down")) {
            types.check(R.id.sit_down);
        }
        else if (current.getType().equals("take_out")) {
            types.check(R.id.take_out);
        }
        else {
            types.check(R.id.delivery);
        }

        getTabHost().setCurrentTab(1);
    }
};

private Runnable longTask=new Runnable() {
    public void run() {
        for (int i=progress;
             i<10000 && isActive.get();
             i+=200) {
            doSomeLongWork(200);
        }

        if (isActive.get()) {
            runOnUiThread(new Runnable() {
                public void run() {
                    setProgressBarVisibility(false);
                    progress=0;
                }
            });
        }
    }
};
```

```
        });
    }
};

class RestaurantAdapter extends ArrayAdapter<Restaurant> {
    RestaurantAdapter() {
        super(LunchList.this, R.layout.row, model);
    }

    public View getView(int position, View convertView,
                        ViewGroup parent) {
        View row=convertView;
        RestaurantHolder holder=null;

        if (row==null) {
            LayoutInflater inflater=getLayoutInflater();

            row=inflater.inflate(R.layout.row, parent, false);
            holder=new RestaurantHolder(row);
            row.setTag(holder);
        }
        else {
            holder=(RestaurantHolder)row.getTag();
        }

        holder.populateFrom(model.get(position));

        return(row);
    }

    static class RestaurantHolder {
        private TextView name=null;
        private TextView address=null;
        private ImageView icon=null;
        private View row=null;

        RestaurantHolder(View row) {
            this.row=row;

            name=(TextView)row.findViewById(R.id.title);
            address=(TextView)row.findViewById(R.id.address);
            icon=(ImageView)row.findViewById(R.id.icon);
        }

        void populateFrom(Restaurant r) {
            name.setText(r.getName());
            address.setText(r.getAddress());

            if (r.getType().equals("sit_down")) {
                icon.setImageResource(R.drawable.ball_red);
            }
            else if (r.getType().equals("take_out")) {

```

```
    icon.setImageResource(R.drawable.ball_yellow);
}
else {
    icon.setImageResource(R.drawable.ball_green);
}
}
```

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Have the progress position be persisted via `onSaveInstanceState()`. When the activity is started in `onCreate()`, see if the background work was in progress when the activity was shut down (i.e., progress further than 0), and restart the background thread immediately if it was. To test this, you can press `<Ctrl>-<F12>` to simulate opening the keyboard and rotating the screen – by default, this causes your activity to be destroyed and recreated, with `onSaveInstanceState()` called along the way.
- Try moving the pause/resume logic to `onStop()` and `onStart()`.

Further Reading

You can find material on the topics shown in this tutorial in the "Handling Activity Lifecycle Events" chapter of [The Busy Coder's Guide to Android Development](#).

You are also strongly encouraged to read the [class overview for Activity](#) in the [JavaDocs](#).

A Few Good Resources

We have already used many types of resources in the preceding tutorials. After reviewing what we have used so far, we set up an alternate layout for our `LunchList` activity to be used when the activity is in landscape orientation instead of portrait.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the `09-Lifecycle` edition of `LunchList` to use as a starting point.

Step #1: Review our Current Resources

Now that we have completed ten tutorials, this is a good time to recap what resources we have been using along the way. Right now, `LunchList` has:

- Seven icons in `LunchList/res/drawable/`, all PNGs
- Two XML files in `LunchList/res/layout/`, representing the main `LunchList` UI and the definition of each row
- One XML file in `LunchList/res/menu/`, containing our option menu definition
- The system-created `strings.xml` file in `LunchList/res/values/`

Step #2: Create a Landscape Layout

In the emulator, with `LunchList` running and showing the details form, press `<ctrl>-<F12>`. This simulates opening and closing the keyboard, causing the screen to rotate to landscape and portrait, respectively. Our current layout is not very good in landscape orientation:

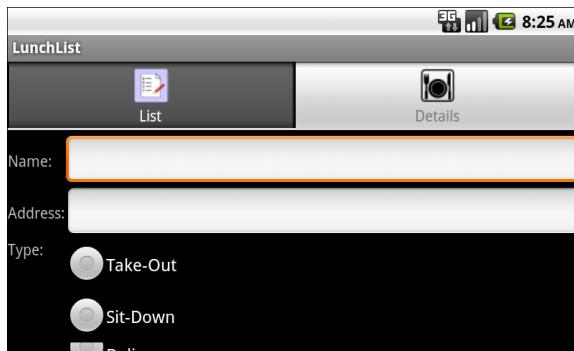


Figure 20. The LunchList in landscape orientation

So, let us come up with an alternative layout that will work better.

First, create a `LunchList/res/layout-land/` directory in your project. This will hold layout files that we wish to use when the device (or emulator) is in the landscape orientation.

Then, copy `LunchList/res/layout/main.xml` into `LunchList/res/layout-land/`, so we can start with the same layout we were using for portrait mode.

Then, change the layout to look like this:

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

```
    android:layout_height="wrap_content"
  />
<FrameLayout android:id="@+id/tabcontent"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
  >
  <LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
  >
    <ListView android:id="@+id/restaurants"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
    />
  </LinearLayout>
<TableLayout android:id="@+id/details"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:stretchColumns="1,3"
    android:paddingTop="4px"
  >
  <TableRow>
    <TextView
      android:text="Name:"
      android:paddingRight="2px"
    />
    <EditText
      android:id="@+id/name"
      android:maxLength="140sp"
    />
    <TextView
      android:text="Address:"
      android:paddingLeft="2px"
      android:paddingRight="2px"
    />
    <EditText
      android:id="@+id/addr"
      android:maxLength="140sp"
    />
  </TableRow>
  <TableRow>
    <TextView android:text="Type:" />
    <RadioGroup android:id="@+id/types">
      <RadioButton android:id="@+id/take_out"
        android:text="Take-Out"
      />
      <RadioButton android:id="@+id/sit_down"
        android:text="Sit-Down"
      />
      <RadioButton android:id="@+id/delivery"
        android:text="Delivery"
      />
    </RadioGroup>
    <TextView
```

```
        android:text="Notes:"
        android:paddingLeft="2px"
    />
    <LinearLayout
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:orientation="vertical"
    >
        <EditText android:id="@+id/notes"
            android:singleLine="false"
            android:gravity="top"
            android:lines="3"
            android:scrollHorizontally="false"
            android:maxLines="3"
            android:maxLength="140sp"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
        />
        <Button android:id="@+id/save"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="Save"
        />
    </LinearLayout>
</TableRow>
</TableLayout>
</FrameLayout>
</LinearLayout>
</TabHost>
```

In this revised layout, we:

- Switched to four columns in our table, with columns #1 and #3 as stretchable
- Put the name and address labels and fields on the same row
- Put the type, notes, and Save button on the same row, with the notes and Save button stacked via a `LinearLayout`
- Made the notes three lines instead of two, since we have the room
- Fixed the maximum width of the `EditText` widgets to 140 scaled pixels (`sp`), so they do not automatically grow outlandishly large if we type a lot
- Added a bit of padding in places to make the placement of the labels and fields look a bit better

If you rebuild and reinstall the application, then run it in landscape mode, you will see a form that looks like this:

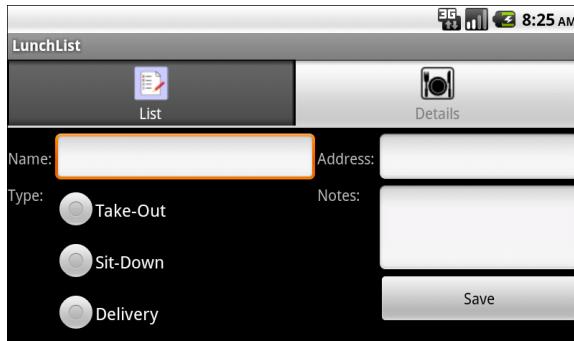


Figure 21. The LunchList in landscape orientation, revised

Note that we did not create a `LunchList/res/layout-land/` edition of our row layout (`row.xml`). Android, upon not finding one in `LunchList/res/layout-land/`, will fall back to the one in `LunchList/res/layout/`. Since we do not really need our row to change, we can leave it as-is.

Note that when you change the screen orientation, your existing restaurants will vanish. That is because we are not persisting them anywhere, and rotating the screen by default destroys and recreates the activity. These issues will be addressed in later tutorials.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Find some other icons to use and create a `LunchList/res/drawable-land` directory with the replacement icons, using the same names as found in `LunchList/res/drawable`. See if exposing the keyboard swaps the icons as well as the layouts.
- Change the text of the labels in our main layout file to be string resources. You will need to add those values to `LunchList/res/values/strings.xml` and reference them in `LunchList/res/layout/main.xml`.

- Use `onSaveInstanceState()` to save the current contents of the details form, and restore those contents in `onCreate()` if an instance state is available (e.g., after the screen was rotated). Note how this does not cover the list – you will still lose all existing restaurants on a rotation event. However, in a later tutorial, we will move that data to the database, which will solve that problem.

Further Reading

You can learn more about resource sets, particularly with respect to UI impacts, in the "Working with Resources" chapter of [The Busy Coder's Guide to Android Development](#).

You will also find "Table 2" in the [Alternate Resources section](#) of the Android developer guide to be of great use for determining the priority of different resource set suffixes.

The Restaurant Store

In this tutorial, we will create a database and table for holding our restaurant data and switch from our `ArrayAdapter` to a `CursorAdapter`, to make use of that database. This will allow our restaurants to persist from run to run of `LunchList`.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 10-Resources edition of `LunchList` to use as a starting point.

Step #1: Create a Stub SQLiteOpenHelper

First, we need to be able to define what our database name is, what the schema is for the table for our restaurants, etc. That is best wrapped up in a `SQLiteOpenHelper` implementation.

So, create `LunchList/src/apt/tutorial/RestaurantHelper.java`, and enter in the following code:

```
package apt.tutorial;  
  
import android.content.Context;  
import android.database.SQLException;
```

```
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteQueryBuilder;

class RestaurantHelper extends SQLiteOpenHelper {
    private static final String DATABASE_NAME="lunchlist.db";
    private static final int SCHEMA_VERSION=1;

    public RestaurantHelper(Context context) {
        super(context, DATABASE_NAME, null, SCHEMA_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {

    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    }
}
```

This says that our database name is `lunchlist.db`, we are using the first version of the schema...and not much else. However, the project should still compile cleanly after adding this class.

Step #2: Manage our Schema

Next, we need to flesh out the `onCreate()` and `onUpgrade()` methods in `RestaurantHelper`, to actually create the schema we want.

To do this, add an import for `android.database.Cursor` and use the following implementation of `onCreate()`:

```
@Override
public void onCreate(SQLiteDatabase db) {
    db.execSQL("CREATE TABLE restaurants (_id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT, address TEXT, type TEXT, notes TEXT);");
}
```

Here, we are simply executing a SQL statement to create a `restaurants` table with a particular schema.

For `onUpgrade()`, there is nothing we really need to do now, since this method will not be executed until we have at least two schema versions. So

far, we barely have our first schema version. So, just put a comment to that effect in `onUpgrade()`, perhaps something like this:

```
@Override  
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {  
    // no-op, since will not be called until 2nd schema  
    // version exists  
}
```

In a production system, of course, we would want to make a temporary table, copy our current data to it, fix up the real table's schema, then migrate the data back.

Step #3: Remove Extraneous Code from LunchList

With our menu and thread samples behind us, we can get rid of our option menu and simplify the code. Get rid of the following items from your implementation of `LunchList`:

- The `isActive` and `progress` data members
- The call to `requestWindowFeature()` in `onCreate()`
- The implementations of `onPause()`, `onResume()`, `onCreateOptionsMenu()`, and `onOptionsItemSelected()`
- The `startWork()` and `doSomeLongWork()` methods, along with the `longTask` Runnable

Step #4: Get Access to the Helper

We will be using `RestaurantHelper` as our bridge to the database. Hence, `LunchList` will need a `RestaurantHelper`, to retrieve existing restaurants and add new ones.

In order to really use the database, though, we need to open and close access to it from `LunchList`.

First, create a `RestaurantHelper` data member named `helper`.

Then, in `onCreate()` in `LunchList`, after the call to `setContentView()`, initialize `RestaurantHelper` like this:

```
helper=new RestaurantHelper(this);
```

Finally, implement `onDestroy()` on `LunchList` as follows:

```
@Override  
public void onDestroy() {  
    super.onDestroy();  
  
    helper.close();  
}
```

All we do in `onDestroy()`, besides chain to the superclass, is close the helper we opened in `onCreate()`. This will close the underlying SQLite database as well.

Step #5: Save a Restaurant to the Database

We are going to be replacing our restaurant object model (and its associated `ArrayList`) with the database and a `Cursor` representing the roster of restaurants. This will involve adding some more logic to `RestaurantHelper` to aid in this process, while also starting to use it from `LunchList`.

First, add an `import` statement for `android.content.ContentValues` to `RestaurantHelper`.

Then, implement `insert()` on `RestaurantHelper` as follows:

```
public void insert(String name, String address,  
                  String type, String notes) {  
    ContentValues cv=new ContentValues();  
  
    cv.put("name", name);  
    cv.put("address", address);  
    cv.put("type", type);  
    cv.put("notes", notes);  
  
    getWritableDatabase().insert("restaurants", "name", cv);  
}
```

With this code, we pour the individual pieces of a restaurant (e.g., its name) into a `ContentValues` and tell the `SQLiteDatabase` to insert it into the database. We call `getWritableDatabase()` to get at the `SQLiteDatabase`. Our helper will automatically open the database in write mode if it has not already been opened by the helper before.

Finally, we need to actually call `insert()` at the appropriate time. Right now, our Save button adds a restaurant to our `RestaurantAdapter` – now, we need it to persist the restaurant to the database. So, modify the `onSave` object in `LunchList` to look like this:

```
private View.OnClickListener onSave=new View.OnClickListener() {
    public void onClick(View v) {
        String type=null;

        switch (types.getCheckedRadioButtonId()) {
            case R.id.sit_down:
                type="sit_down";
                break;
            case R.id.take_out:
                type="take_out";
                break;
            case R.id.delivery:
                type="delivery";
                break;
        }

        helper.insert(name.getText().toString(),
                     address.getText().toString(), type,
                     notes.getText().toString());
    }
};
```

We simply get the four pieces of data from their respective widgets and call `insert()`.

Step #6: Get the List of Restaurants from the Database

This puts restaurants into the database. Presumably, it would be useful to get them back out sometime. Hence, we need some logic that can query the database and return a `Cursor` with columnar data from our `restaurant` table. A `Cursor` in Android is much like a cursor in other database access libraries

- it is an encapsulation of the result set of the query, plus the query that was used to create it.

To do this, add the following method to `RestaurantHelper`:

```
public Cursor getAll() {
    return(getReadableDatabase()
        .rawQuery("SELECT _id, name, address, type, notes FROM restaurants
ORDER BY name",
                  null));
}
```

Here, we get access to the underlying `SQLiteDatabase` (opening it in read mode if it is not already open) and call `rawQuery()`, passing in a suitable query string to retrieve all restaurants, sorted by name.

We will also need to have some way to get the individual pieces of data out of the `Cursor` (e.g., `name`). To that end, add a few getter-style methods to `RestaurantHelper` that will retrieve the proper columns from a `Cursor` positioned on the desired row:

```
public String getName(Cursor c) {
    return(c.getString(1));
}

public String getAddress(Cursor c) {
    return(c.getString(2));
}

public String getType(Cursor c) {
    return(c.getString(3));
}

public String getNotes(Cursor c) {
    return(c.getString(4));
}
```

Step #7: Change our Adapter and Wrapper

Of course, our existing `RestaurantAdapter` extends `ArrayAdapter` and cannot use a `Cursor` very effectively. So, we need to change our `RestaurantAdapter` into something that can use a `Cursor`...such as a `CursorAdapter`. Just as an `ArrayAdapter` creates a `View` for every needed item in an array or `List`, `CursorAdapter` creates a `View` for every needed row in a `Cursor`.

A `CursorAdapter` does not use `getView()`, but rather `bindView()` and `newView()`. The `newView()` method handles the case where we need to inflate a new row; `bindView()` is when we are recycling an existing row. So, our current `getView()` logic needs to be split between `bindView()` and `newView()`.

Replace our existing `RestaurantAdapter` implementation in `LunchList` with the following:

```
class RestaurantAdapter extends CursorAdapter {
    RestaurantAdapter(Cursor c) {
        super(LunchList.this, c);
    }

    @Override
    public void bindView(View row, Context ctxt,
                         Cursor c) {
        RestaurantHolder holder=(RestaurantHolder)row.getTag();

        holder.populateFrom(c, helper);
    }

    @Override
    public View newView(Context ctxt, Cursor c,
                        ViewGroup parent) {
        LayoutInflater inflater=getLayoutInflater();
        View row=inflater.inflate(R.layout.row, parent, false);
        RestaurantHolder holder=new RestaurantHolder(row);

        row.setTag(holder);

        return(row);
    }
}
```

Then, you need to make use of this refined adapter, by changing the `model` in `LunchList` from an `ArrayList` to a `Cursor`. After you have changed that data

member, replace the current `onCreate()` code that populates our `RestaurantAdapter` with the following:

```
model=helper.getAll();
startManagingCursor(model);
adapter=new RestaurantAdapter(model);
list.setAdapter(adapter);
```

After getting the Cursor from `getAll()`, we call `startManagingCursor()`, so Android will deal with refreshing its contents if the activity is paused and resumed. Then, we hand the Cursor off to the `RestaurantAdapter`.

Also, you will need to import `android.content.Context` and `android.widget.CursorAdapter` in `LunchList`.

Then, we need to update `RestaurantHolder` to work with `Cursor` objects rather than a restaurant directly. Replace the existing implementation with the following:

```
static class RestaurantHolder {
    private TextView name=null;
    private TextView address=null;
    private ImageView icon=null;
    private View row=null;

    RestaurantHolder(View row) {
        this.row=row;

        name=(TextView)row.findViewById(R.id.title);
        address=(TextView)row.findViewById(R.id.address);
        icon=(ImageView)row.findViewById(R.id.icon);
    }

    void populateFrom(Cursor c, RestaurantHelper helper) {
        name.setText(helper.getName(c));
        address.setText(helper.getAddress(c));

        if (helper.getType(c).equals("sit_down")) {
            icon.setImageResource(R.drawable.ball_red);
        }
        else if (helper.getType(c).equals("take_out")) {
            icon.setImageResource(R.drawable.ball_yellow);
        }
        else {
            icon.setImageResource(R.drawable.ball_green);
        }
    }
}
```

{}

Step #8: Clean Up Lingering ArrayList References

Since we changed our model in `LunchList` from an `ArrayList` to a `Cursor`, anything that still assumes an `ArrayList` will not work.

Notably, the `onListClick` listener object tries to obtain a restaurant from the `ArrayList`. Now, we need to move the `Cursor` to the appropriate position and get a restaurant from that. So, modify `onListClick` to use the `Cursor` and the property getter methods on `RestaurantHelper` instead:

```
private AdapterView.OnItemClickListener onListClick=new
AdapterView.OnItemClickListener() {
    public void onItemClick(AdapterView<?> parent,
                        View view, int position,
                        long id) {
        model.moveToPosition(position);
        name.setText(helper.getName(model));
        address.setText(helper.getAddress(model));
        notes.setText(helper.getNotes(model));

        if (helper.getType(model).equals("sit_down")) {
            types.check(R.id.sit_down);
        }
        else if (helper.getType(model).equals("take_out")) {
            types.check(R.id.take_out);
        }
        else {
            types.check(R.id.delivery);
        }

        getTabHost().setCurrentTab(1);
    };
}
```

At this point, you can recompile and reinstall your application. If you try using it, it will launch and you can save restaurants to the database. However, you will find that the list of restaurants will not update unless you exit and restart the `LunchList` activity.

Step #9: Refresh Our List

The reason the list does not update is because neither the `Cursor` nor the `CursorAdapter` realize that the database contents have changed when we save our restaurant. To resolve this, add `model.requery();` immediately after the call to `insert()` in the `onSave` object in `LunchList`. This causes the `Cursor` to reload its contents from the database, which in turn will cause the `CursorAdapter` to redisplay the list.

Rebuild and reinstall the application and try it out. You should have all the functionality you had before, with the added benefit of restaurants living from run to run of `LunchList`.

Here is an implementation of `LunchList` that incorporates all of the changes shown in this tutorial:

```
package apt.tutorial;

import android.app.TabActivity;
import android.content.Context;
import android.database.Cursor;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.view.LayoutInflater;
import android.widget.AdapterView;
import android.widget.CursorAdapter;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ImageView;
import android.widget.ListView;
import android.widget.RadioGroup;
import android.widget.TabHost;
import android.widget.TextView;

public class LunchList extends TabActivity {
    Cursor model=null;
    RestaurantAdapter adapter=null;
    EditText name=null;
    EditText address=null;
    EditText notes=null;
    RadioGroup types=null;
    RestaurantHelper helper=null;

    @Override
    public void onCreate(Bundle savedInstanceState) {
```

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

helper=new RestaurantHelper(this);

name=(EditText)findViewById(R.id.name);
address=(EditText)findViewById(R.id.addr);
notes=(EditText)findViewById(R.id.notes);
types=(RadioGroup)findViewById(R.id.types);

Button save=(Button)findViewById(R.id.save);

save.setOnClickListener(onSave);

ListView list=(ListView)findViewById(R.id.restaurants);

model=helper.getAll();
startManagingCursor(model);
adapter=new RestaurantAdapter(model);
list.setAdapter(adapter);

TabHost.TabSpec spec=getTabHost().newTabSpec("tag1");

spec.setContent(R.id.restaurants);
spec.setIndicator("List", getResources()
    .getDrawable(R.drawable.list));
getTabHost().addTab(spec);

spec=getTabHost().newTabSpec("tag2");
spec.setContent(R.id.details);
spec.setIndicator("Details", getResources()
    .getDrawable(R.drawable.restaurant));
getTabHost().addTab(spec);

getTabHost().setCurrentTab(0);

list.setOnItemClickListener(onListClick);
}

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

    helper.close();
}

private View.OnClickListener onSave=new View.OnClickListener() {
    public void onClick(View v) {
        String type=null;

        switch (types.getCheckedRadioButtonId()) {
            case R.id.sit_down:
                type="sit_down";
                break;
        }
    }
}
```

```
        case R.id.take_out:
            type="take_out";
            break;
        case R.id.delivery:
            type="delivery";
            break;
    }

    helper.insert(name.getText().toString(),
                  address.getText().toString(), type,
                  notes.getText().toString());
    model.requery();
}
};

private AdapterView.OnItemClickListener onListClick=new
AdapterView.OnItemClickListener() {
    public void onItemClick(AdapterView<?> parent,
                           View view, int position,
                           long id) {
        model.moveToPosition(position);
        name.setText(helper.getName(model));
        address.setText(helper.getAddress(model));
        notes.setText(helper.getNotes(model));

        if (helper.getType(model).equals("sit_down")) {
            types.check(R.id.sit_down);
        }
        else if (helper.getType(model).equals("take_out")) {
            types.check(R.id.take_out);
        }
        else {
            types.check(R.id.delivery);
        }

        getTabHost().setCurrentTab(1);
    }
};

class RestaurantAdapter extends CursorAdapter {
    RestaurantAdapter(Cursor c) {
        super(LunchList.this, c);
    }

    @Override
    public void bindView(View row, Context ctxt,
                        Cursor c) {
        RestaurantHolder holder=(RestaurantHolder)row.getTag();

        holder.populateFrom(c, helper);
    }

    @Override
    public View newView(Context ctxt, Cursor c,
```

```
    ViewGroup parent) {
    LayoutInflater inflater=getLayoutInflater();
    View row=inflater.inflate(R.layout.row, parent, false);
    RestaurantHolder holder=new RestaurantHolder(row);

    row.setTag(holder);

    return(row);
}

static class RestaurantHolder {
    private TextView name=null;
    private TextView address=null;
    private ImageView icon=null;
    private View row=null;

    RestaurantHolder(View row) {
        this.row=row;

        name=(TextView)row.findViewById(R.id.title);
        address=(TextView)row.findViewById(R.id.address);
        icon=(ImageView)row.findViewById(R.id.icon);
    }

    void populateFrom(Cursor c, RestaurantHelper helper) {
        name.setText(helper.getName(c));
        address.setText(helper.getAddress(c));

        if (helper.getType(c).equals("sit_down")) {
            icon.setImageResource(R.drawable.ball_red);
        }
        else if (helper.getType(c).equals("take_out")) {
            icon.setImageResource(R.drawable.ball_yellow);
        }
        else {
            icon.setImageResource(R.drawable.ball_green);
        }
    }
}
```

Similarly, here is a full implementation of `RestaurantHelper` that contains the modifications from this tutorial:

```
package apt.tutorial;

import android.content.Context;
import android.content.ContentValues;
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteOpenHelper;
```

```
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteQueryBuilder;

class RestaurantHelper extends SQLiteOpenHelper {
    private static final String DATABASE_NAME="lunchlist.db";
    private static final int SCHEMA_VERSION=1;

    public RestaurantHelper(Context context) {
        super(context, DATABASE_NAME, null, SCHEMA_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL("CREATE TABLE restaurants (_id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT, address TEXT, type TEXT, notes TEXT');");
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // no-op, since will not be called until 2nd schema
        // version exists
    }

    public Cursor getAll() {
        return(getReadableDatabase()
            .rawQuery("SELECT _id, name, address, type, notes FROM restaurants
ORDER BY name",
            null));
    }

    public void insert(String name, String address,
                      String type, String notes) {
        ContentValues cv=new ContentValues();

        cv.put("name", name);
        cv.put("address", address);
        cv.put("type", type);
        cv.put("notes", notes);

        getWritableDatabase().insert("restaurants", "name", cv);
    }

    public String getName(Cursor c) {
        return(c.getString(1));
    }

    public String getAddress(Cursor c) {
        return(c.getString(2));
    }

    public String getType(Cursor c) {
        return(c.getString(3));
    }
}
```

```
public String getNotes(Cursor c) {  
    return(c.getString(4));  
}
```

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Download the database off the emulator (or device) and examine it using a SQLite client program. You can use `adb pull` to download `/data/data/apt.tutorial/databases/lunchlist.db`, or use Eclipse or DDMS to browse the emulator graphically to retrieve the same file.
- Use `adb shell` and the `sqlite3` program built into the emulator to examine the database in the emulator itself, without downloading it.

Further Reading

You can learn more about how Android and SQLite work together in the "Managing and Accessing Local Databases" chapter of [The Busy Coder's Guide to Android Development](#).

However, if you are looking for more general documentation on SQLite itself, such as its particular flavor of SQL, you will want to use the [SQLite site](#), or perhaps [The Definitive Guide to SQLite](#).

Getting More Active

In this tutorial, we will add support for both creating new restaurants and editing ones that were previously entered. Along the way, we will get rid of our tabs, splitting the application into two activities: one for the list, and one for the detail form.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the `11-Database` edition of `LunchList` to use as a starting point.

Also, for this specific tutorial, since there is a lot of cutting and pasting, you may wish to save off a copy of your current work before starting in on the modifications, so you can clip code from the original and paste it where it is needed.

Step #1: Create a Stub Activity

The first thing we need to do is create an activity to serve as our detail form. In a flash of inspiration, let's call it `DetailForm`. So, create a `LunchList/src/apt/tutorial/DetailForm.java` file with the following content:

```
package apt.tutorial;

import android.app.Activity;
import android.os.Bundle;

public class DetailForm extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //    setContentView(R.layout.main);
    }
}
```

This is just a stub activity, except it has the `setContentView()` line commented out. That is because we do not want to use `main.xml`, as that is the layout for `LunchList`. Since we do not have another layout ready yet, we can just comment out the line. As we will see, this is perfectly legal, but it means the activity will have no UI.

Step #2: Launch the Stub Activity on List Click

Now, we need to arrange to display this activity when the user clicks on a `LunchList` list item, instead of flipping to the original detail form tab in `LunchList`.

First, we need to add `DetailForm` to the `AndroidManifest.xml` file, so it is recognized by the system as being an available activity. Change the manifest to look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="apt.tutorial"
    android:versionCode="1"
    android:versionName="1.0">
    <application android:label="@string/app_name">
        <activity android:name=".LunchList"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".DetailForm">
        </activity>
```

```
</application>  
</manifest>
```

Notice the second `<activity>` element, referencing the `DetailForm` class. Also note that it does not need an `<intent-filter>`, since we will be launching it ourselves rather than expecting the system to launch it for us.

Then, we need to start this activity when the list item is clicked. That is handled by our `onListClick` listener object. So, replace our current implementation with the following:

```
private AdapterView.OnItemClickListener onListClick=new  
AdapterView.OnItemClickListener() {  
    public void onItemClick(AdapterView<?> parent,  
                           View view, int position,  
                           long id) {  
        Intent i=new Intent(LunchList.this, DetailForm.class);  
  
        startActivity(i);  
    }  
};
```

Here we create an `Intent` that points to our `DetailForm` and call `startActivity()` on that `Intent`. You will need to add an import for `android.content.Intent` to `LunchList`.

At this point, you should be able to recompile and reinstall the application. If you run it and click on an item in the list, it will open up the empty `DetailForm`. From there, you can click the BACK button to return to the main `LunchList` activity.

Step #3: Move the Detail Form UI

Now, the shredding begins – we need to start moving our detail form smarts out of `LunchList` and its layout to `DetailForm`.

First, create a `LunchList/res/layout/detail_form.xml`, using the detail form from `LunchList/res/layout/main.xml` as a basis:

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:stretchColumns="1"
    >
    <TableRow>
        <TextView android:text="Name: " />
        <EditText android:id="@+id/name" />
    </TableRow>
    <TableRow>
        <TextView android:text="Address: " />
        <EditText android:id="@+id/addr" />
    </TableRow>
    <TableRow>
        <TextView android:text="Type: " />
        <RadioGroup android:id="@+id/types">
            <RadioButton android:id="@+id/take_out"
                android:text="Take-Out"
            />
            <RadioButton android:id="@+id/sit_down"
                android:text="Sit-Down"
            />
            <RadioButton android:id="@+id/delivery"
                android:text="Delivery"
            />
        </RadioGroup>
    </TableRow>
    <TableRow>
        <TextView android:text="Notes: " />
        <EditText android:id="@+id/notes"
            android:singleLine="false"
            android:gravity="top"
            android:lines="2"
            android:scrollHorizontally="false"
            android:maxLines="2"
            android:maxWidth="200sp"
        />
    </TableRow>
    <Button android:id="@+id/save"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Save"
    />
</TableLayout>
```

This is just the detail form turned into its own standalone layout file.

Next, un-comment the `setContentView()` call in `onCreate()` in `DetailForm` and have it load this layout:

```
setContentView(R.layout.detail_form);
```

Then, we need to add all our logic for accessing the various form widgets, plus an `onSave` listener for our Save button, plus all necessary imports.

Set the import list for `DetailForm` to be:

```
import android.app.Activity;
import android.database.Cursor;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.RadioGroup;
import android.widget.TextView;
```

Then, add the following data members to the `DetailForm` class:

```
EditText name=null;
EditText address=null;
EditText notes=null;
RadioGroup types=null;
RestaurantHelper helper=null;
```

Then, copy the widget finders and stuff from `onCreate()` in `LunchList` into the same spot in `DetailForm`:

```
helper=new RestaurantHelper(this);

name=(EditText)findViewById(R.id.name);
address=(EditText)findViewById(R.id.addr);
notes=(EditText)findViewById(R.id.notes);
types=(RadioGroup)findViewById(R.id.types);

Button save=(Button)findViewById(R.id.save);

save.setOnClickListener(onSave);
```

Finally, add the `onSave` listener object with a subset of the implementation from `LunchList`:

```
private View.OnClickListener onSave=new View.OnClickListener() {
    public void onClick(View v) {
        String type=null;

        switch (types.getCheckedRadioButtonId()) {
            case R.id.sit_down:
                type="sit_down";
```

```
        break;
    case R.id.take_out:
        type="take_out";
        break;
    case R.id.delivery:
        type="delivery";
        break;
    }
};
```

Step #4: Clean Up the Original UI

Now we need to clean up `LunchList` and its layout to reflect the fact that we moved much of the logic over to `DetailForm`.

First, get rid of the tabs and the detail form from `LunchList/res/layout/main.xml`, and alter the `ListView`'s `android:id` to something suitable for `ListActivity`, leaving us with:

```
<?xml version="1.0" encoding="utf-8"?>
<ListView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/list"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
/>
```

Next, delete `LunchList/res/layout_land/main.xml`, as we will revisit landscape layouts in a later tutorial.

At present, `LunchList` extends `TabActivity`, which is no longer what we need. Change it to extend `ListActivity` instead, adding an import for `android.app.ListActivity`.

Finally, get rid of the code from `onCreate()` that sets up the tabs and the Save button, since they are no longer needed. Also, you no longer need to find the `ListView` widget, since you can call `setListAdapter()` on the `ListActivity` to associate your `RestaurantAdapter` with the `ListActivity`'s `ListView`. The resulting `onCreate()` implementation should look like:

```
@Override  
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.main);  
  
    helper=new RestaurantHelper(this);  
  
    name=(EditText)findViewById(R.id.name);  
    address=(EditText)findViewById(R.id.addr);  
    notes=(EditText)findViewById(R.id.notes);  
    types=(RadioGroup)findViewById(R.id.types);  
  
    model=helper.getAll();  
    startManagingCursor(model);  
    adapter=new RestaurantAdapter(model);  
    setListAdapter(adapter);  
}
```

Step #5: Pass the Restaurant _ID

Now, let's step back a bit and think about what we are trying to achieve.

We want to be able to use `DetailForm` for both adding new restaurants and editing an existing restaurant. `DetailForm` needs to be able to tell those two scenarios apart. Also, `DetailForm` needs to know which item is to be edited.

To achieve this, we will pass an "extra" in our `Intent` that launches `DetailForm`, containing the ID (`_id` column) of the restaurant to edit. We will use this if the `DetailForm` was launched by clicking on an existing restaurant. If `DetailForm` receives an `Intent` lacking our "extra", it will know to add a new restaurant.

First, we need to define a name for this "extra", so add the following data member to `LunchList`:

```
public final static String ID_EXTRA="apt.tutorial._ID";
```

We use the `apt.tutorial` namespace to ensure our "extra" name will not collide with any names perhaps used by the Android system.

Next, convert the `onListClick` object to an `onListItemClick()` method (available to us on `ListActivity`) and have it add this "extra" to the Intent it uses to start the `DetailForm`:

```
public void onListItemClick(ListView list, View view,
                           int position, long id) {
    Intent i=new Intent(LunchList.this, DetailForm.class);
    i.putExtra(ID_EXTRA, String.valueOf(id));
    startActivity(i);
}
```

The `_id` of the restaurant happens to be provided to us as the fourth parameter to `onListItemClick()`. We turn it into a `String` because `DetailForm` will want it in `String` format, as we will see shortly.

Next, add the following data member to `DetailForm`:

```
String restaurantId=null;
```

This will be `null` if we are adding a new restaurant or the string form of the ID if we are editing an existing restaurant.

Finally, add the following line to the end of `onCreate()` in `DetailForm`:

```
restaurantId=getIntent().getStringExtra(LunchList.ID_EXTRA);
```

This will pull out our "extra", or leave `restaurantId` as `null` if there is no such "extra".

Step #6: Load the Restaurant Into the Form

In the case where we are editing an existing restaurant, we need to load that restaurant from the database, then load it into the `DetailForm`.

Since we created a `RestaurantHelper` in `onCreate()`, we need to close it again, so add an `onDestroy()` implementation to `DetailForm` as follows:

```
@Override  
public void onDestroy() {  
    super.onDestroy();  
  
    helper.close();  
}
```

Now that we have a handle to the database, we need to load a restaurant given its ID. So, add the following method to `RestaurantHelper`:

```
public Cursor getById(String id) {  
    String[] args={id};  
  
    return(getReadableDatabase()  
        .rawQuery("SELECT _id, name, address, type, notes FROM restaurants  
WHERE _ID=?",  
        args));  
}
```

Then, add the following lines to the bottom of `onCreate()` in `DetailForm`, to load in the specified restaurant into the form if its ID was specified in the Intent:

```
if (restaurantId!=null) {  
    load();  
}
```

The code snippet above references a `load()` method, which we need to add to `DetailForm`, based off of code originally in `LunchList`:

```
private void load() {  
    Cursor c=helper.getById(restaurantId);  
  
    c.moveToFirst();  
    name.setText(helper.getName(c));  
    address.setText(helper.getAddress(c));  
    notes.setText(helper.getNotes(c));  
  
    if (helper.getType(c).equals("sit_down")) {  
        types.check(R.id.sit_down);  
    }  
    else if (helper.getType(c).equals("take_out")) {  
        types.check(R.id.take_out);  
    }  
    else {  
        types.check(R.id.delivery);  
    }  
}
```

```
c.close();  
}
```

Step #7: Add an "Add" Menu Option

We have most of the logic in place to edit existing restaurants. However, we still need to add a menu item for adding a new restaurant.

To do this, change `LunchList/res/menu/option.xml` to replace the existing options with one for add:

```
<?xml version="1.0" encoding="utf-8"?>  
<menu xmlns:android="http://schemas.android.com/apk/res/android">  
    <item android:id="@+id/add"  
        android:title="Add"  
        android:icon="@drawable/ic_menu_add"  
    />  
</menu>
```

Note that the `add` menu item references an icon supplied by Android. You can find a copy of this icon in your Android SDK. Go to the directory where you installed the SDK, and go into the `platforms/` directory inside of it. Then, go into the directory for some version of Android (e.g., `android-8/`), and into `data/res/drawable-mdpi/`. You will find `ic_menu_add.png` in there.

Now that we have the menu option, we need to adjust our menu handling to match. Restore our older implementation of `onCreateOptionsMenu()` to `LunchList`:

```
public boolean onCreateOptionsMenu(Menu menu) {  
    new MenuInflater(this).inflate(R.menu.option, menu);  
  
    return(super.onCreateOptionsMenu(menu));  
}
```

Then, add an `onOptionsItemSelected()` implementation in `LunchList` with the following:

```
public boolean onOptionsItemSelected(MenuItem item) {  
    if (item.getItemId()==R.id.add) {  
        startActivity(new Intent(LunchList.this, DetailForm.class));  
    }  
}
```

```
        return(true);
    }

    return(super.onOptionsItemSelected(item));
}
```

Here, we launch the `DetailForm` activity without our "extra", signalling to `DetailForm` that it is to add a new restaurant. You will need imports again for `android.view.Menu`, `android.view.MenuInflater`, and `android.view.MenuItem`.

Step #8: Detail Form Supports Add and Edit

Last, but certainly not least, we need to have `DetailForm` properly do useful work when the Save button is clicked. Specifically, we need to either insert or update the database. It would also be nice if we dismissed the `DetailForm` at that point and returned to the main `LunchList` activity.

To accomplish this, we first need to add an `update()` method to `RestaurantHelper` that can perform a database update:

```
public void update(String id, String name, String address,
                   String type, String notes) {
    ContentValues cv=new ContentValues();
    String[] args={id};

    cv.put("name", name);
    cv.put("address", address);
    cv.put("type", type);
    cv.put("notes", notes);

    getWritableDatabase().update("restaurants", cv, "_ID=?",
                               args);
}
```

Then, we need to adjust our `onSave` listener object in `DetailForm` to call the right method (`save()` or `update()`) and `finish()` our activity:

```
private View.OnClickListener onSave=new View.OnClickListener() {
    public void onClick(View v) {
        String type=null;

        switch (types.getCheckedRadioButtonId()) {
            case R.id.sit_down:
```

```
        type="sit_down";
        break;
    case R.id.take_out:
        type="take_out";
        break;
    case R.id.delivery:
        type="delivery";
        break;
    }

    if (restaurantId==null) {
        helper.insert(name.getText().toString(),
                      address.getText().toString(), type,
                      notes.getText().toString());
    }
    else {
        helper.update(restaurantId, name.getText().toString(),
                      address.getText().toString(), type,
                      notes.getText().toString());
    }

    finish();
};
```

At this point, you should be able to recompile and reinstall the application. When you first bring up the application, it will no longer show the tabs:

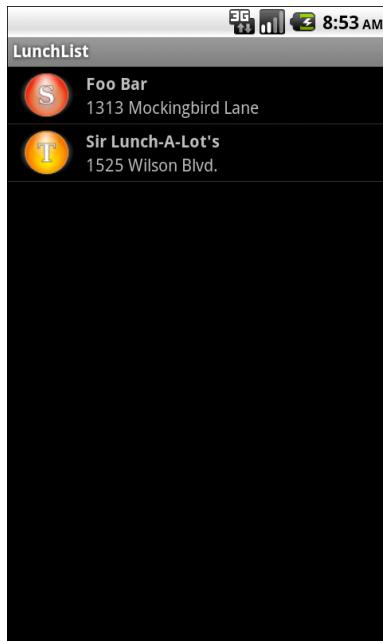


Figure 22. The new-and-improved LunchList

However, it will have an "add" menu option:

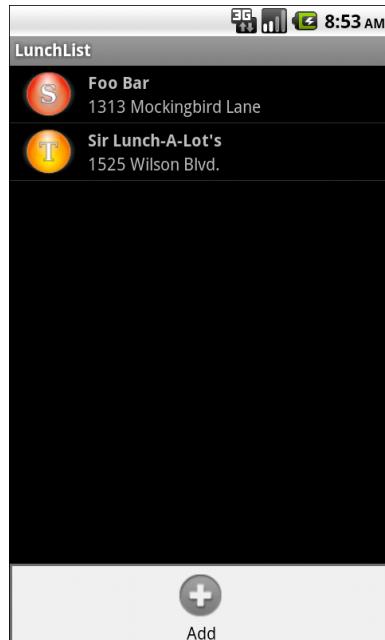


Figure 23. The LunchList option menu, with Add

If you choose the "add" menu option, it will bring up a blank DetailForm:



Figure 24. The DetailForm activity

If you fill out the form and click Save, it will return you to the `LunchList` and immediately shows the new restaurant (courtesy of our using a managed Cursor in `LunchList`):

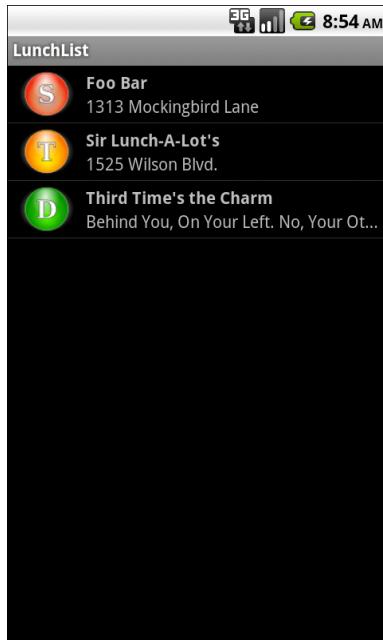


Figure 25. The LunchList with an added Restaurant

If you click an existing restaurant, it will bring up the `DetailForm` for that object:

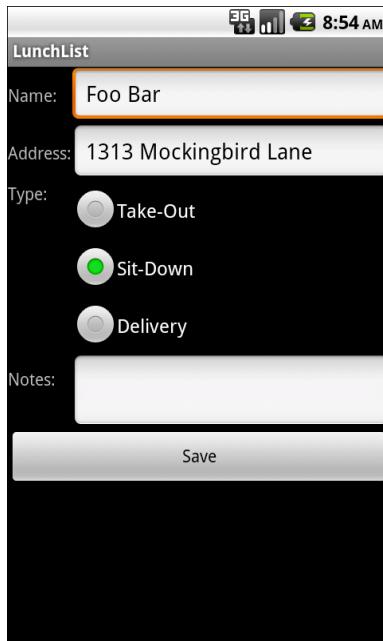


Figure 26. The DetailForm on an existing Restaurant

Making changes and clicking Save will update the database and list:

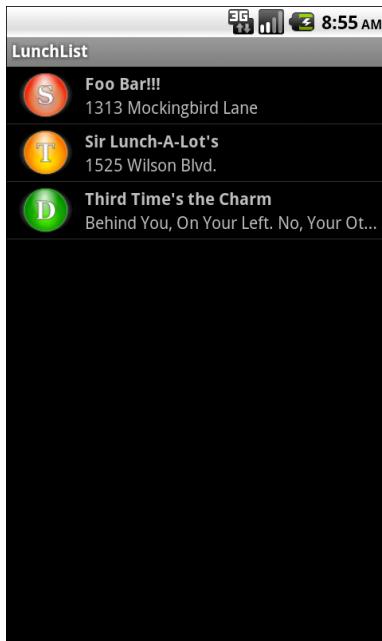


Figure 27. The LunchList with an edited Restaurant

Here is one implementation of `LunchList` that incorporates all of this tutorial's changes:

```
package apt.tutorial;

import android.app.ListActivity;
import android.content.Context;
import android.content.Intent;
import android.database.Cursor;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.view.ViewGroup;
import android.view.LayoutInflater;
import android.widget.AdapterView;
import android.widget.CursorAdapter;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ImageView;
import android.widget.ListView;
import android.widget.RadioGroup;
import android.widget.TabHost;
import android.widget.TextView;
```

```
public class LunchList extends ListActivity {
    public final static String ID_EXTRA="apt.tutorial._ID";
    Cursor model=null;
    RestaurantAdapter adapter=null;
    EditText name=null;
    EditText address=null;
    EditText notes=null;
    RadioGroup types=null;
    RestaurantHelper helper=null;

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

        helper=new RestaurantHelper(this);

        name=(EditText)findViewById(R.id.name);
        address=(EditText)findViewById(R.id.addr);
        notes=(EditText)findViewById(R.id.notes);
        types=(RadioGroup)findViewById(R.id.types);

        model=helper.getAll();
        startManagingCursor(model);
        adapter=new RestaurantAdapter(model);
        setListAdapter(adapter);
    }

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

        helper.close();
    }

    @Override
    public void onListItemClick(ListView list, View view,
                               int position, long id) {
        Intent i=new Intent(LunchList.this, DetailForm.class);

        i.putExtra(ID_EXTRA, String.valueOf(id));
        startActivity(i);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        new MenuInflater(this).inflate(R.menu.option, menu);

        return(super.onCreateOptionsMenu(menu));
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
```

```
if (item.getItemId()==R.id.add) {
    startActivity(new Intent(LunchList.this, DetailForm.class));
    return(true);
}

return(super.onOptionsItemSelected(item));
}

private View.OnClickListener onSave=new View.OnClickListener() {
    public void onClick(View v) {
        String type=null;

        switch (types.getCheckedRadioButtonId()) {
            case R.id.sit_down:
                type="sit_down";
                break;
            case R.id.take_out:
                type="take_out";
                break;
            case R.id.delivery:
                type="delivery";
                break;
        }

        helper.insert(name.getText().toString(),
                     address.getText().toString(), type,
                     notes.getText().toString());
        model.requery();
    }
};

class RestaurantAdapter extends CursorAdapter {
    RestaurantAdapter(Cursor c) {
        super(LunchList.this, c);
    }

    @Override
    public void bindView(View row, Context ctxt,
                         Cursor c) {
        RestaurantHolder holder=(RestaurantHolder)row.getTag();

        holder.populateFrom(c, helper);
    }

    @Override
    public View newView(Context ctxt, Cursor c,
                        ViewGroup parent) {
        LayoutInflater inflater=getLayoutInflater();
        View row=inflater.inflate(R.layout.row, parent, false);
        RestaurantHolder holder=new RestaurantHolder(row);

        row.setTag(holder);
    }
};
```

```
        return(row);
    }

static class RestaurantHolder {
    private TextView name=null;
    private TextView address=null;
    private ImageView icon=null;
    private View row=null;

    RestaurantHolder(View row) {
        this.row=row;

        name=(TextView)row.findViewById(R.id.title);
        address=(TextView)row.findViewById(R.id.address);
        icon=(ImageView)row.findViewById(R.id.icon);
    }

    void populateFrom(Cursor c, RestaurantHelper helper) {
        name.setText(helper.getName(c));
        address.setText(helper.getAddress(c));

        if (helper.getType(c).equals("sit_down")) {
            icon.setImageResource(R.drawable.ball_red);
        }
        else if (helper.getType(c).equals("take_out")) {
            icon.setImageResource(R.drawable.ball_yellow);
        }
        else {
            icon.setImageResource(R.drawable.ball_green);
        }
    }
}
```

Here is one implementation of `DetailForm` that works with the revised `LunchList`:

```
package apt.tutorial;

import android.app.Activity;
import android.database.Cursor;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.RadioGroup;
import android.widget.TextView;

public class DetailForm extends Activity {
    EditText name=null;
    EditText address=null;
```

```
EditText notes=null;
RadioGroup types=null;
RestaurantHelper helper=null;
String restaurantId=null;

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

    helper=new RestaurantHelper(this);

    name=(EditText)findViewById(R.id.name);
    address=(EditText)findViewById(R.id.addr);
    notes=(EditText)findViewById(R.id.notes);
    types=(RadioGroup)findViewById(R.id.types);

    Button save=(Button)findViewById(R.id.save);
    save.setOnClickListener(onSave);

    restaurantId=getIntent().getStringExtra(LunchList.ID_EXTRA);

    if (restaurantId!=null) {
        load();
    }
}

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

    helper.close();
}

private void load() {
    Cursor c=helper.getViewById(restaurantId);

    c.moveToFirst();
    name.setText(helper.getName(c));
    address.setText(helper.getAddress(c));
    notes.setText(helper.getNotes(c));

    if (helper.getType(c).equals("sit_down")) {
        types.check(R.id.sit_down);
    }
    else if (helper.getType(c).equals("take_out")) {
        types.check(R.id.take_out);
    }
    else {
        types.check(R.id.delivery);
    }

    c.close();
}
```

```
}

private View.OnClickListener onSave=new View.OnClickListener() {
    public void onClick(View v) {
        String type=null;

        switch (types.getCheckedRadioButtonId()) {
            case R.id.sit_down:
                type="sit_down";
                break;
            case R.id.take_out:
                type="take_out";
                break;
            case R.id.delivery:
                type="delivery";
                break;
        }

        if (restaurantId==null) {
            helper.insert(name.getText().toString(),
                         address.getText().toString(), type,
                         notes.getText().toString());
        }
        else {
            helper.update(restaurantId, name.getText().toString(),
                         address.getText().toString(), type,
                         notes.getText().toString());
        }

        finish();
    }
};
```

And, here is an implementation of RestaurantHelper with the changes needed by DetailForm:

```
package apt.tutorial;

import android.content.Context;
import android.content.ContentValues;
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteQueryBuilder;

class RestaurantHelper extends SQLiteOpenHelper {
    private static final String DATABASE_NAME="lunchlist.db";
    private static final int SCHEMA_VERSION=1;

    public RestaurantHelper(Context context) {
```

```
    super(context, DATABASE_NAME, null, SCHEMA_VERSION);
}

@Override
public void onCreate(SQLiteDatabase db) {
    db.execSQL("CREATE TABLE restaurants (_id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT, address TEXT, type TEXT, notes TEXT);");
}

@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    // no-op, since will not be called until 2nd schema
    // version exists
}

public Cursor getAll() {
    return(getReadableDatabase()
        .rawQuery("SELECT _id, name, address, type, notes FROM restaurants
ORDER BY name",
                  null));
}

public Cursor getById(String id) {
    String[] args={id};

    return(getReadableDatabase()
        .rawQuery("SELECT _id, name, address, type, notes FROM restaurants
WHERE _ID=?",
                  args));
}

public void insert(String name, String address,
                   String type, String notes) {
    ContentValues cv=new ContentValues();

    cv.put("name", name);
    cv.put("address", address);
    cv.put("type", type);
    cv.put("notes", notes);

    getWritableDatabase().insert("restaurants", "name", cv);
}

public void update(String id, String name, String address,
                   String type, String notes) {
    ContentValues cv=new ContentValues();
    String[] args={id};

    cv.put("name", name);
    cv.put("address", address);
    cv.put("type", type);
    cv.put("notes", notes);

    getWritableDatabase().update("restaurants", cv, "_ID=?",

```

```
        args);  
    }  
  
    public String getName(Cursor c) {  
        return(c.getString(1));  
    }  
  
    public String getAddress(Cursor c) {  
        return(c.getString(2));  
    }  
  
    public String getType(Cursor c) {  
        return(c.getString(3));  
    }  
  
    public String getNotes(Cursor c) {  
        return(c.getString(4));  
    }  
}
```

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Have the database hold a URL for the restaurant's Web site. Update the UI to collect this address in the detail form. Launch that URL via `startActivity()` via an option menu choice from the restaurant list, so you can view the restaurant's Web site.
- Add an option menu to delete a restaurant. Raise an `AlertDialog` to confirm that the user wants the restaurant deleted. Delete it from the database and refresh the list if the user confirms the deletion.

Further Reading

You can read up on having multiple activities in your application, or linking to activities supplied by others, in the "Launching Activities and Sub-Activities" chapter of [The Busy Coder's Guide to Android Development](#).

What's Your Preference?

In this tutorial, we will add a preference setting for the sort order of the restaurant list. To do this, we will create a `PreferenceScreen` definition in XML, load that into a `PreferenceActivity`, connect that activity to the application, and finally actually use the preference to control the sort order.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 12-Activities edition of `LunchList` to use as a starting point.

Step #1: Define the Preference XML

First, add a `LunchList/res/xml/preferences.xml` file as follows:

```
<PreferenceScreen
    xmlns:android="http://schemas.android.com/apk/res/android">
    <ListPreference
        android:key="sort_order"
        android:title="Sort Order"
        android:summary="Choose the order the list uses"
        android:entries="@array/sort_names"
        android:entryValues="@array/sort_clauses"
        android:dialogTitle="Choose a sort order" />
</PreferenceScreen>
```

This sets up a single-item `PreferenceScreen`. Note that it references two string arrays, one for the display labels of the sort-order selection list, and one for the values actually stored in the `SharedPreferences`.

So, to define those string arrays, add a `LunchList/res/values/arrays.xml` file with the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="sort_names">
        <item>By Name, Ascending</item>
        <item>By Name, Descending</item>
        <item>By Type</item>
        <item>By Address, Ascending</item>
        <item>By Address, Descending</item>
    </string-array>
    <string-array name="sort_clauses">
        <item>name ASC</item>
        <item>name DESC</item>
        <item>type, name ASC</item>
        <item>address ASC</item>
        <item>address DESC</item>
    </string-array>
</resources>
```

Note we are saying that the value stored in the `SharedPreferences` will actually be an `ORDER BY` clause for use in our `SQL` query. This is a convenient trick, though it does tend to make the system a bit more fragile – if we change our column names, we might have to change our preferences to match and deal with older invalid preference values.

Step #2: Create the Preference Activity

Next, we need to create a `PreferenceActivity` that will actually use these preferences. To do this, add a `PreferenceActivity` implementation, stored as `LunchList/src/apt/tutorial/EditPreferences.java`:

```
package apt.tutorial;

import android.app.Activity;
import android.os.Bundle;
import android.preference.PreferenceActivity;

public class EditPreferences extends PreferenceActivity {
```

```
@Override  
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
  
    addPreferencesFromResource(R.xml.preferences);  
}  
}
```

We also need to update `AndroidManifest.xml` to reference this activity, so we can launch it later:

```
<?xml version="1.0" encoding="utf-8"?>  
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    package="apt.tutorial"  
    android:versionCode="1"  
    android:versionName="1.0">  
    <application android:label="@string/app_name">  
        <activity android:name=".LunchList"  
            android:label="@string/app_name">  
            <intent-filter>  
                <action android:name="android.intent.action.MAIN" />  
                <category android:name="android.intent.category.LAUNCHER" />  
            </intent-filter>  
        </activity>  
        <activity android:name=".DetailForm">  
        </activity>  
        <activity android:name=".EditPreferences">  
        </activity>  
    </application>  
</manifest>
```

Step #3: Connect the Preference Activity to the Option Menu

Now, we can add a menu option to launch the `EditPreferences` activity.

We need to add another `<item>` to our `LunchList/res/menu/option.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>  
<menu xmlns:android="http://schemas.android.com/apk/res/android">  
    <item android:id="@+id/add"  
        android:title="Add"  
        android:icon="@drawable/ic_menu_add"  
    />  
    <item android:id="@+id/prefs"  
        android:title="Settings"  
        android:icon="@drawable/ic_menu_preferences"
```

```
/>  
</menu>
```

We reference an `ic_menu_preferences.png` file, which you can obtain from the same directory where you got `ic_menu_add.png`.

Of course, if we modify the menu XML, we also need to modify the `LunchList` implementation of `onOptionsItemSelected()` to match, so replace the current implementation with the following:

```
@Override  
public boolean onOptionsItemSelected(MenuItem item) {  
    if (item.getItemId()==R.id.add) {  
        startActivity(new Intent(LunchList.this, DetailForm.class));  
  
        return(true);  
    }  
    else if (item.getItemId()==R.id.prefs) {  
        startActivity(new Intent(this, EditPreferences.class));  
  
        return(true);  
    }  
  
    return(super.onOptionsItemSelected(item));  
}
```

All we are doing is starting up our `EditPreferences` activity.

If you recompile and reinstall the application, you will see our new menu option:

What's Your Preference?

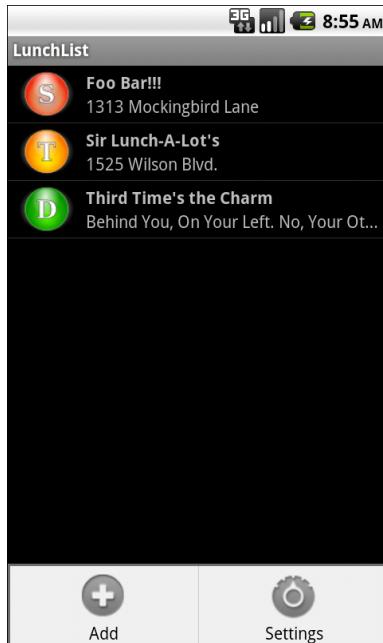


Figure 28. The LunchList with the new menu option

And if you choose that menu option, you will get the `EditPreferences` activity:

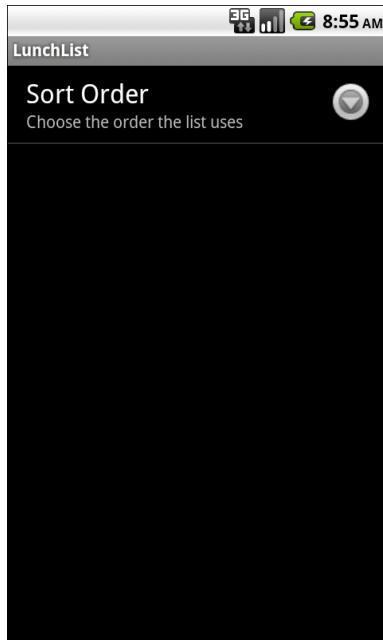


Figure 29. The preferences activity

Clicking the Sort Order item will bring up a selection list of available sort orders:

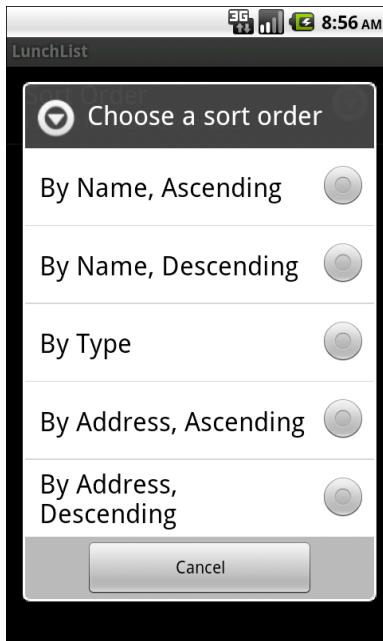


Figure 30. The available sort orders

Of course, none of this is actually having any effect on the sort order itself, which we will address in the next section.

Step #4: Apply the Sort Order on Startup

Now, given that the user has chosen a sort order, we need to actually use it. First, we can apply it when the application starts up – the next section will handle changing the sort order after the user changes the preference value.

First, the `getAll()` method on `RestaurantHelper` needs to take a sort order as a parameter, rather than apply one of its own. So, change that method as follows:

```
public Cursor getAll(String orderBy) {
    return(getReadableDatabase()
        .rawQuery("SELECT _id, name, address, type, notes FROM restaurants
ORDER BY "+orderBy,
        null));
}
```

Then, we need to get our hands on our `SharedPreferences` instance. Add imports to `LunchList` for `android.content.SharedPreferences` and `android.preference.PreferenceManager`, along with a `SharedPreferences` data member named `prefs`.

Next, add this line near the top of `onCreate()` in `LunchList`, to initialize `prefs` to be the `SharedPreferences` our preference activity uses:

```
prefs=PreferenceManager.getDefaultSharedPreferences(this);
```

Finally, change the call to `getAll()` to use the `SharedPreferences`:

```
model=helper.getAll(prefs.getString("sort_order", "name"));
```

Here, we use `name` as the default value, so if the user has not specified a sort order yet, the sort order will be by name.

Now, if you recompile and reinstall the application, then set a sort order preference, you can see that preference take effect if you exit and reopen the application.

Step #5: Listen for Preference Changes

That works, but users will get annoyed if they have to exit the application just to get their preference choice to take effect. To change the sort order on the fly, we first need to know when they change the sort order.

`SharedPreferences` has the notion of a preference listener object, to be notified on such changes. To take advantage of this, add the following line at the end of `onCreate()` in `LunchList`:

```
prefs.registerOnSharedPreferenceChangeListener(prefListener);
```

This snippet refers to a `prefListener` object, so add the following code to `LunchList` to create a stub implementation of that object:

```
private SharedPreferences.OnSharedPreferenceChangeListener prefListener=
    new SharedPreferences.OnSharedPreferenceChangeListener() {
        public void onSharedPreferenceChanged(SharedPreferences sharedPrefs, String
key) {
            if (key.equals("sort_order")) {
            }
        }
   };
```

All we are doing right now is watching for our specific preference of interest (`sort_order`), though we are not actually taking advantage of the changed value.

Step #6: Re-Apply the Sort Order on Changes

Finally, we actually need to change the sort order. For simple lists like this, the easiest way to accomplish this is to get a fresh `Cursor` representing our list (from `getAll()` on `RestaurantHelper`) with the proper sort order, and use the new `Cursor` instead of the old one.

First, pull some of the list-population logic out of `onCreate()`, by implementing an `initList()` method as follows:

```
private void initList() {
    if (model!=null) {
        stopManagingCursor(model);
        model.close();
    }

    model=helper.getAll(prefs.getString("sort_order", "name"));
    startManagingCursor(model);
    adapter=new RestaurantAdapter(model);
    setListAdapter(adapter);
}
```

Note that we call `stopManagingCursor()` so Android will ignore the old `Cursor`, then we close it, before we get and apply the new `Cursor`. Of course, we only do those things if there is an old `Cursor`.

The `onCreate()` method needs to change to take advantage of `initList()`:

```
@Override  
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.main);  
  
    helper=new RestaurantHelper(this);  
    prefs=PreferenceManager.getDefaultSharedPreferences(this);  
  
    name=(EditText)findViewById(R.id.name);  
    address=(EditText)findViewById(R.id.addr);  
    notes=(EditText)findViewById(R.id.notes);  
    types=(RadioGroup)findViewById(R.id.types);  
  
    initList();  
    prefs.registerOnSharedPreferenceChangeListener(prefListener);  
}
```

Also, we can call `initList()` from `prefListener`:

```
private SharedPreferences.OnSharedPreferenceChangeListener prefListener=  
    new SharedPreferences.OnSharedPreferenceChangeListener() {  
        public void onSharedPreferenceChanged(SharedPreferences sharedPrefs,  
                                              String key) {  
            if (key.equals("sort_order")) {  
                initList();  
            }  
        }  
   };
```

At this point, if you recompile and reinstall the application, you should see the sort order change immediately as you change the order via the preferences.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Add a preference for the default type of restaurant (e.g., take-out). Use that preference in detail forms when creating a new restaurant.
- Add an option menu to the detail form activity and have it be able to start the preference activity the way we did from the option menu for the list.
- Rather than use preferences, store the preference values in a JSON file that you read in at startup and re-read in `onResume()` (to find out

about changes). This means you will need to create your own preference UI, rather than rely upon the one created by the preference XML.

Further Reading

Learn more about setting up preference XML files and reading shared preferences in the "Using Preferences" chapter of [The Busy Coder's Guide to Android Development](#).

Turn, Turn, Turn

In this tutorial, we will make our application somewhat more intelligent about screen rotations, ensuring that partially-entered restaurant information remains intact even after the screen rotates.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 13-Prefs edition of `LunchList` to use as a starting point.

Step #1: Add a Stub `onSaveInstanceState()`

Since we are not holding onto network connections or other things that cannot be stored in a `Bundle`, we can use `onSaveInstanceState()` to track our state as the screen is rotated.

To that end, add a stub implementation of `onSaveInstanceState()` to `DetailForm` as follows:

```
@Override  
public void onSaveInstanceState(Bundle state) {  
    super.onSaveInstanceState(state);  
}
```

Step #2: Pour the Form Into the Bundle

Now, fill in the details of `onSaveInstanceState()`, putting our widget contents into the supplied `Bundle`:

```
@Override  
public void onSaveInstanceState(Bundle state) {  
    super.onSaveInstanceState(state);  
  
    state.putString("name", name.getText().toString());  
    state.putString("address", address.getText().toString());  
    state.putString("notes", notes.getText().toString());  
    state.putInt("type", types.getCheckedRadioButtonId());  
}
```

Step #3: Repopulate the Form

Next, we need to make use of that saved state. We could do this in `onCreate()`, if the passed-in `Bundle` is non-null. However, it is usually easier just to override `onRestoreInstanceState()`. This is called only when there is state to restore, supplying the `Bundle` with your state. So, add an implementation of `onRestoreInstanceState()` to `DetailForm`:

```
@Override  
public void onRestoreInstanceState(Bundle state) {  
    super.onRestoreInstanceState(state);  
  
    name.setText(state.getString("name"));  
    address.setText(state.getString("address"));  
    notes.setText(state.getString("notes"));  
    types.check(state.getInt("type"));  
}
```

At this point, you can recompile and reinstall the application. Use `<ctrl>-
<F12>` to simulate rotating the screen of your emulator. If you do this after making changes (but not saving) on the `DetailForm`, you will see those changes survive the rotation.

Step #4: Fix Up the Landscape Detail Form

As you tested the work from the previous section, you no doubt noticed that the `DetailForm` layout is not well-suited for landscape – the notes text

area is chopped off and the Save button is missing. To fix this, we need to create a `LunchList/res/layout-land/detail_form.xml` file, derived from our original, but set up to take advantage of the whitespace to the right of the radio buttons:

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:stretchColumns="1,3"
    >
    <TableRow>
        <TextView android:text="Name:" />
        <EditText android:id="@+id/name"
            android:layout_span="3"
        />
    </TableRow>
    <TableRow>
        <TextView android:text="Address:" />
        <EditText android:id="@+id/addr"
            android:layout_span="3"
        />
    </TableRow>
    <TableRow>
        <TextView android:text="Type:" />
        <RadioGroup android:id="@+id/types">
            <RadioButton android:id="@+id/take_out"
                android:text="Take-Out"
            />
            <RadioButton android:id="@+id/sit_down"
                android:text="Sit-Down"
            />
            <RadioButton android:id="@+id/delivery"
                android:text="Delivery"
            />
        </RadioGroup>
        <TextView android:text="Notes:" />
        <LinearLayout
            android:layout_width="fill_parent"
            android:layout_height="fill_parent"
            android:orientation="vertical"
            >
            <EditText android:id="@+id/notes"
                android:singleline="false"
                android:gravity="top"
                android:lines="4"
                android:scrollHorizontally="false"
                android:maxLines="4"
                android:maxLength="140sp"
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
            />
            <Button android:id="@+id/save"

```

```
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Save"
  />
</LinearLayout>
</TableRow>
</TableLayout>
```

Now, if you recompile and reinstall the application, you should see a better landscape rendition of the detail form:

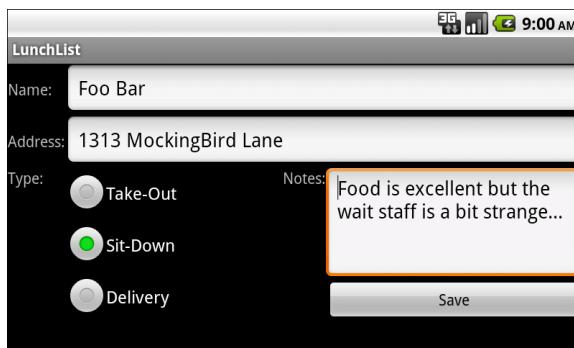


Figure 31. The new landscape layout

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Try switching to `onRetainNonConfigurationInstance()` instead of `onSaveInstanceState()`.
- Try commenting out `onSaveInstanceState()`. Does the activity still retain its instance state? Why or why not?
- Have the application automatically rotate based on physical orientation instead of keyboard position. Hint: find a place to apply `android:screenOrientation = "sensor"`.

Further Reading

Additional coverage of screen rotations and how to control what happens during them can be found in the "Handling Rotation" chapter of [The Busy Coder's Guide to Android Development](#).

PART II – Network Application Tutorials

Raising (Something Like) a Tweet

In this tutorial, we will experiment with the Apache HttpClient library in Android, using it to post a status update to our [identi.ca](#) account. The identi.ca service, powered by StatusNet, is a microblogging service similar to Twitter. In fact, this book used to profile a Twitter client. However, Twitter's new authentication models, while good for Twitter, are a bit too complicated to implement for this book.

Over the course of several tutorials, we will focus on microblog-related functionality in an application we will call `Patchy` (which used to be short for "Patchy: Another Twitter Client? Heck, Yeah!").

Step-By-Step Instructions

This tutorial starts a new application, independent from the `LunchList` application developed in the preceding tutorials.

Step #1: Set Up an Identi.ca Account

Twitter is a micro-blogging service that you may have heard about. On the other hand, you may not have an identi.ca account.

If you have such an account and do not mind using it for creating a Patchy application, feel free to stick with it. Otherwise, create an identi.ca account that you can use for your experimentation. Visit [the identi.ca Web site](#) and sign up. Note that identi.ca will send a confirmation email to you, and you must click the link on that email in order to be able to work with the REST API.

Step #2: Create a Stub Application and Activity

Using `Eclipse` or `android create project`, make a project named `Patchy` with a stub activity named `apt.tutorial.two.Patchy`. The generated activity class should resemble the following:

```
package apt.tutorial.two;

import android.app.Activity;
import android.os.Bundle;

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

Step #3: Create a Layout

Next, we need a layout for use with the `Patchy` activity. At the outset, we want:

- A field for your identi.ca user name
- A field for your identi.ca password, needed to use much of the identi.ca REST API
- A multi-line field for a status update you want to publish
- A Send button that will send the status update

Here is a simple `TableLayout` containing those items, one that looks a bit like the `DetailForm` from the `LunchList` tutorial earlier in this book:

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:stretchColumns="1"
    >
    <TableRow>
        <TextView android:text="User Name:" />
        <EditText android:id="@+id/user" />
    </TableRow>
    <TableRow>
        <TextView android:text="Password:" />
        <EditText android:id="@+id/password"
            android:password="true"
        />
    </TableRow>
    <TableRow>
        <TextView android:text="Status:" />
        <EditText android:id="@+id/status"
            android:singleLine="false"
            android:gravity="top"
            android:lines="5"
            android:scrollHorizontally="false"
            android:maxLines="5"
            android:maxWidth="200sp"
        />
    </TableRow>
    <Button android:id="@+id/send"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Send"
    />
</TableLayout>
```

If you compile and reinstall this application, you will see the layout in action:

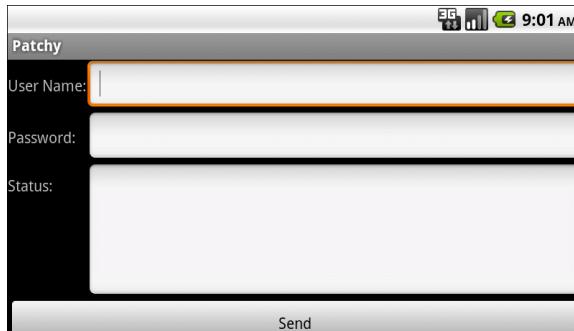


Figure 32. The initial Patchy layout, in landscape

Step #4: Listen for Send Actions

Next, we need to get control when the user clicks the Send button. This involves finding the Send button in our layout and attaching a listener to it.

Replace the stock implementation of `Patchy` with the following:

```
package apt.tutorial.two;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;

public class Patchy extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        Button send=(Button)findViewById(R.id.send);

        send.setOnClickListener(onSend);
    }

    private View.OnClickListener onSend=new View.OnClickListener() {
        public void onClick(View v) {
        }
    };
}
```

Step #5: Make the Status Post Request

With all that done, we can actually invoke the identi.ca REST API. This is somewhat tedious, for a few reasons:

- identi.ca requires HTTP Basic authentication to pass over the user name and password, rather than embedding them as values in, say, an HTTP POST request. HttpClient does not handle preemptive HTTP authentication very well.
- HTTP Basic authentication requires Base64 encoding, and Android (before 2.2) lacks a publicly-visible Base64 encoder.

Here is what you need to do to get past all of this and make `Patchy` work:

First, find a suitable Base64 encoder, such as [this public domain one](#). Put the `Base64` class in your project in `Patchy/src/apt/tutorial/two/` and add the matching package line to the top of the file, so it is available to your `Patchy` implementation. Note that Android 2.2 has a `Base64` encoder class, though that has not been tried as a part of this tutorial.

Next, replace your existing `Patchy` implementation with the following:

```
package apt.tutorial.two;

import android.app.Activity;
import android.app.AlertDialog;
import android.os.Bundle;
import android.view.View;
import android.util.Log;
import android.widget.Button;
import android.widget.EditText;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import org.apache.http.NameValuePair;
import org.apache.http.HttpVersion;
import org.apache.http.client.ResponseHandler;
import org.apache.http.client.HttpClient;
import org.apache.http.client.entity.UrlEncodedFormEntity;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.impl.client.BasicResponseHandler;
import org.apache.http.impl.client.DefaultHttpClient;
import org.apache.http.message.BasicNameValuePair;
import org.apache.http.protocol.HTTP;
```

```
import org.json.JSONObject;

public class Patchy extends Activity {
    private DefaultHttpClient client=null;
    private EditText user=null;
    private EditText password=null;
    private EditText status=null;

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

        user=(EditText)findViewById(R.id.user);
        password=(EditText)findViewById(R.id.password);
        status=(EditText)findViewById(R.id.status);

        Button send=(Button)findViewById(R.id.send);
        send.setOnClickListener(onSend);

        client=new DefaultHttpClient();
    }

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

        client.getConnectionManager().shutdown();
    }

    private String getCredentials() {
        String u=user.getText().toString();
        String p=password.getText().toString();

        return(Base64.encodeBytes((u+":"+p).getBytes()));
    }

    private void updateStatus() {
        try {
            String s=status.getText().toString();

            HttpPost post=new HttpPost("https://identi.ca/api/statuses/update.json");

            post.addHeader("Authorization",
                "Basic "+getCredentials());

            List<NameValuePair> form=new ArrayList<NameValuePair>();
            form.add(new BasicNameValuePair("status", s));

            post.setEntity(new UrlEncodedFormEntity(form, HTTP.UTF_8));
            ResponseHandler<String> responseHandler=new BasicResponseHandler();
        }
    }
}
```

```
String responseBody=client.execute(post, responseHandler);
JSONObject response=new JSONObject(responseBody);
}
catch (Throwable t) {
    Log.e("Patchy", "Exception in updateStatus()", t);
    goBlooeey(t);
}
}

private void goBlooeey(Throwable t) {
    AlertDialog.Builder builder=new AlertDialog.Builder(this);

    builder
        .setTitle("Exception!")
        .setMessage(t.toString())
        .setPositiveButton("OK", null)
        .show();
}

private View.OnClickListener onSend=new View.OnClickListener() {
    public void onClick(View v) {
        updateStatus();
    }
};
}
```

Here, we:

- Set up an `DefaultHttpClient` instance for accessing the Apache `HttpClient` engine
- Get access to our `EditText` widgets from the layout
- On a Send button click, call `updateStatus()`
- In `updateStatus()`, we create an `HttpPost` object to represent the request, fill in the authentication credentials, fill in the status as a form element, turn the whole thing into a valid HTTP POST operation, execute it, and parse the response as a JSON object

If things work, at present, we do nothing to update the activity; if an `Exception` is raised, we log it to the Android log and raise an `AlertDialog`.

You also need to add the `INTERNET` permission to your `AndroidManifest.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="apt.tutorial.two"
    android:versionCode="1"
    android:versionName="1.0">
    <uses-permission android:name="android.permission.INTERNET" />
    <application android:label="@string/app_name">
        <activity android:name=".Patchy"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

If you recompile and reinstall the application, you should now be able to update your identi.ca status by filling in the fields and clicking the Send button.

You will notice that we are making this HTTP request on the main application thread. That is not a good practice – it can freeze the user interface, and if the request takes too long, Android will display the dreaded "application not responding" dialog box. However, we will address that problem in the next tutorial (or in an Extra Credit item, if you prefer).

If you work in a facility that requires a proxy server, your emulator may be unhappy (as might Android devices operating on WiFi behind the proxy). You may need to add the following snippet of code to work past that, adjusting the parameters to suit your office's setup:

```
Properties systemSettings=System.getProperties();
systemSettings.put("http.proxyHost", "your.proxy.host.here");
systemSettings.put("http.proxyPort", "8080"); // use actual proxy port
```

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Process the post request in a background thread and display a `Toast` when the request is complete and successful.

- Add logic to check for the 140-character message limit before sending the request. If the message is too long, display an `AlertDialog`.
- Make the `AlertDialog` from the previous point obsolete by adding the `android:maxLength` attribute to the `EditText` for the message, to constrain it to 140 characters.
- Add in TinyURL support by adding a field for a URL to append to the end of the message, then using the TinyURL REST API (`http://tinyurl.com/api-create.php?url = ...` returns the shortened URL as a response) to generate the shortened URL, then attach it to the message before making the request.
- Experiment with the Android 2.2 `AndroidHttpClient` class, as an alternative to `DefaultHttpClient`.

Further Reading

Additional examples of interacting with `HttpClient` from Android can be found in the "Communicating via the Internet" chapter of [The Busy Coder's Guide to Android Development](#).

However, the definitive resource for `HttpClient` is the Apache site for [HttpClient](#) itself. In particular, their [samples](#) show a number of techniques we will not be using here.

Opening a JAR

Writing our own identi.ca API seems silly, considering that there are so many of them available as open source, including three for Java. In this tutorial, we will replace our `HttpClient` with [JTwitter](#), an LGPL Twitter/identi.ca API that works with Android.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the `15-HttpClient` edition of `Patchy` to use as a starting point.

Step #1: Obtain the JTwitter JAR

The current official JTwitter distribution appears to drop support for identi.ca. Hence, this tutorial will have you use a patched version of an identi.ca-compatible JTwitter distribution.

Download the modified [JTwitter JAR](#). Then, put the JAR in the `Patchy/libs/` directory, so it is available to your application. Or, you can obtain the same modified JTwitter JAR from the [tutorial results](#).

NOTE: Eclipse users will also need to add the `jtwitter.jar` file to their build path.

The source code to this modified JTtwitter implementation can be found on the [CommonsWare Web site](#).

Step #2: Switch from HttpClient to JTtwitter

Now, we can get rid of most of our previous `Patchy` implementation, including all of the `HttpClient` code, and replace it with the delightfully simply `JTwitter` API.

Replace the current implementation of `Patchy` with the following:

```
package apt.tutorial.two;

import android.app.Activity;
import android.app.AlertDialog;
import android.os.Bundle;
import android.view.View;
import android.util.Log;
import android.widget.Button;
import android.widget.EditText;
import winterwell.jtwitter.Twitter;

public class Patchy extends Activity {
    private EditText user=null;
    private EditText password=null;
    private EditText status=null;

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

        user=(EditText)findViewById(R.id.user);
        password=(EditText)findViewById(R.id.password);
        status=(EditText)findViewById(R.id.status);

        Button send=(Button)findViewById(R.id.send);

        send.setOnClickListener(onSend);
    }

    private void updateStatus() {
        try {
```

```
Twitter client=new Twitter(user.getText().toString(),
                           password.getText().toString());

client.setAPIRootUrl("https://identi.ca/api");
client.updateStatus(status.getText().toString());
}

catch (Throwable t) {
    Log.e("Patchy", "Exception in updateStatus()", t);
    goBlooeey(t);
}

private void goBlooeey(Throwable t) {
    AlertDialog.Builder builder=new AlertDialog.Builder(this);

builder
    .setTitle("Exception!")
    .setMessage(t.toString())
    .setPositiveButton("OK", null)
    .show();
}

private View.OnClickListener onSend=new View.OnClickListener() {
    public void onClick(View v) {
        updateStatus();
    }
};
```

Besides getting rid of all HttpClient references, we reimplemented updateStatus() to create a Twitter object (with our user name and password), then called its own updateStatus() method with the typed-in status.

If you rebuild and reinstall Patchy, you can once again update your status, just with half of the lines of code.

Step #3: Create Preferences for Account Information

Rather than have the user enter their account information every time they run Patchy, we should store that information in user preferences, so it can be entered only on occasion.

First, we need a preference XML file, so create a new file, Patchy/res/xml/preferences.xml, with the following contents:

```
<PreferenceScreen
    xmlns:android="http://schemas.android.com/apk/res/android">
    <PreferenceCategory android:title="User Account">
        <EditTextPreference
            android:key="user"
            android:title="User Name"
            android:summary="Your identi.ca screen name"
            android:dialogTitle="Enter your identi.ca user name" />
        <EditTextPreference
            android:key="password"
            android:title="Password"
            android:summary="Your identi.ca account password"
            android:password="true"
            android:dialogTitle="Enter your identi.ca password" />
    </PreferenceCategory>
</PreferenceScreen>
```

Note the use of `android:password = "true"` to turn the second one into a password-style preference. Any `EditText` attributes found in the `EditTextPreference` element are passed through to the `EditText` used in the popup dialog.

Then, we need another `EditPreference` activity akin to the one we used in `LunchList` – call it `Patchy/src/apt/tutorial/two/EditPreferences.java`:

```
package apt.tutorial.two;

import android.app.Activity;
import android.os.Bundle;
import android.preference.PreferenceActivity;

public class EditPreferences extends PreferenceActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        addPreferencesFromResource(R.xml.preferences);
    }
}
```

We must not forget to update our `AndroidManifest.xml` file to reference the new activity, so amend yours to look like:

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

```
<uses-permission android:name="android.permission.INTERNET" />
<application android:label="@string/app_name">
    <activity android:name=".Patchy"
              android:label="@string/app_name">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <activity android:name=".EditPreferences">
    </activity>
</application>
</manifest>
```

Then, we want to have an option menu to trigger editing the preferences, so create Patchy/res/menu/option.xml with the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/prefs"
          android:title="Settings"
          android:icon="@drawable/ic_menu_preferences"
    />
</menu>
```

You will need an `ic_menu_preferences.png` drawable resource, perhaps the one you used in the `LunchList` tutorials.

Finally, we need to clone the code from `LunchList` that opens our option menu using the above XML and routes the item click to the `EditPreference` activity. Paste the following methods into `Patchy`:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    new MenuInflater(getApplicationContext())
        .inflate(R.menu.option, menu);

    return(super.onCreateOptionsMenu(menu));
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.prefs) {
        startActivity(new Intent(this, EditPreferences.class));

        return(true);
    }
}
```

```
    return(super.onOptionsItemSelected(item));  
}
```

You will also need to add some imports (`Intent`, `Menu`, `MenuItem`) to allow this to compile.

At this point, you can recompile and reinstall the application and see our preference activity in action, even though the values are not being used yet:

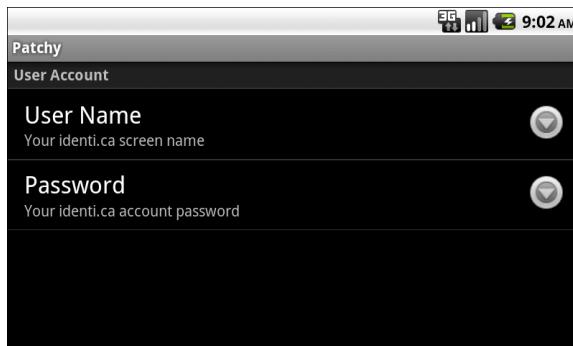


Figure 33. The preference screen for Patchy

Step #4: Use Account Information from Preferences

Now, we can get rid of the extraneous widgets on our layout and use the preferences for our account information. We can even arrange to update our `Twitter` object when the user changes account information.

First, go into `Patchy/res/layout/main.xml` and get rid of the first two `TableRow` elements, leaving you with:

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:stretchColumns="1"  
    >  
    <TableRow>  
        <TextView android:text="Status:" />  
        <EditText android:id="@+id/status"  
            android:singleLine="false"  
            android:gravity="top"  
            android:lines="5"
```

```
    android:scrollHorizontally="false"
    android:maxLines="5"
    android:maxWidth="200sp"
  />
</TableRow>
<Button android:id="@+id/send"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Send"
  />
</TableLayout>
```

Then, eliminate references to the removed widgets from the list of data members and from `onCreate()`.

Next, create a private `SharedPreferences` data member named `prefs` and set it up in `onCreate()`:

```
prefs=PreferenceManager.getDefaultSharedPreferences(this);
prefs.registerOnSharedPreferenceChangeListener(prefListener);
```

The aforementioned code references an `prefListener` object, so define it as follows:

```
private SharedPreferences.OnSharedPreferenceChangeListener prefListener=
  new SharedPreferences.OnSharedPreferenceChangeListener() {
  public void onSharedPreferenceChanged(SharedPreferences sharedPrefs, String
key) {
    if (key.equals("user") || key.equals("password")) {
      resetClient();
    }
  }
};
```

Basically, when the preference changes, we want to reset our Twitter client. This, of course, assumes we have a Twitter client hanging around, so create a private Twitter data member named `client`, and add two methods to work with it as follows:

```
synchronized private Twitter getClient() {
  if (client==null) {
    client=new Twitter(prefs.getString("user", ""),
                      prefs.getString("password", ""));
  }
  client.setAPIRootUrl("https://identi.ca/api");
}
```

```
    }
    return(client);
}

synchronized private void resetClient() {
    client=null;
}
```

Finally, in `updateStatus()`, get rid of the references to the former widgets and use `getClient()` to lazy-create our `Twitter` object, as follows:

```
private void updateStatus() {
    try {
        getClient().updateStatus(status.getText().toString());
    }
    catch (Throwable t) {
        Log.e("Patchy", "Exception in updateStatus()", t);
        goBlooey(t);
    }
}
```

The net is that we will use one `Twitter` object, on our first status update, until the application is closed or the user changes credentials.

If you rebuild and reinstall the application, you should now be able to update your status, yet only enter the status information, using the user name and password from the preferences.

Here is the complete implementation of `Patchy` after completing this tutorial:

```
package apt.tutorial.two;

import android.app.Activity;
import android.app.AlertDialog;
import android.content.Intent;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.preference.PreferenceManager;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.util.Log;
import android.widget.Button;
```

```
import android.widget.EditText;
import winterwell.jtwitter.Twitter;

public class Patchy extends Activity {
    private EditText status=null;
    private Sharedpreferences prefs=null;
    private Twitter client=null;

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

        status=(EditText)findViewById(R.id.status);

        Button send=(Button)findViewById(R.id.send);

        send.setOnClickListener(onSend);

        prefs=PreferenceManager.getDefaultSharedPreferences(this);
        prefs.registerOnSharedPreferenceChangeListener(prefListener);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        new MenuInflater(getApplicationContext())
            .inflate(R.menu.option, menu);

        return(super.onCreateOptionsMenu(menu));
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        if (item.getItemId()==R.id.prefs) {
            startActivity(new Intent(this, EditPreferences.class));

            return(true);
        }

        return(super.onOptionsItemSelected(item));
    }

    synchronized private Twitter getClient() {
        if (client==null) {
            client=new Twitter(prefs.getString("user", ""),
                prefs.getString("password", ""));

            client.setAPIRootUrl("https://identi.ca/api");
        }

        return(client);
    }

    synchronized private void resetClient() {
```

```
    client=null;
}

private void updateStatus() {
    try {
        getClient().updateStatus(status.getText().toString());
    }
    catch (Throwable t) {
        Log.e("Patchy", "Exception in updateStatus()", t);
        goBlooeey(t);
    }
}

private void goBlooeey(Throwable t) {
    AlertDialog.Builder builder=new AlertDialog.Builder(this);

    builder
        .setTitle("Exception!")
        .setMessage(t.toString())
        .setPositiveButton("OK", null)
        .show();
}

private View.OnClickListener onSend=new View.OnClickListener() {
    public void onClick(View v) {
        updateStatus();
    }
};

private SharedPreferences.OnSharedPreferenceChangeListener prefListener=
    new SharedPreferences.OnSharedPreferenceChangeListener() {
    public void onSharedPreferenceChanged(SharedPreferences sharedPrefs, String
key) {
        if (key.equals("user") || key.equals("password")) {
            resetClient();
        }
    }
};
}
```

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Add a checkbox to toggle whether the activity is creating a direct message or a status update. If the checkbox is checked, enable a Spinner containing the list of the user's followers (obtained from `getFollowers()`). For a direct message, use `sendMessage()` on the

`Twitter` object. You may need to "follow" some classmates for everyone to have followers in their accounts to direct message.

- Use the `getStatus()` method on `Twitter` to display the user's current status at the top of the activity when the activity is opened. Re-fetch the status one second after submitting a status update, so the current status shows identi.ca's rendition of your status – a good way to make sure the round-trip of status information is working as you expect.
- Store the password in an encrypted form in the user preferences, so that if somebody hacked a user's phone and got the preference store, they would still need the encryption key to find out the user's account password.

Further Reading

You can learn more about the rules for what third-party code will work with Android in the "Leveraging Java Libraries" chapter of [The Busy Coder's Guide to Android Development](#).

Listening To Your Friends

In this tutorial, we will start watching our friends' timelines, polling them for updates in a background thread managed by a service.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 16-JAR edition of Patchy to use as a starting point.

Step #1: Create a Service Stub

Since we are creating a service to monitor information related to our account, let us create a service named `PostMonitor` (so named because identi.ca supports the original Twitter API). Create a file named `Patchy/src/apt/tutorial/two/PostMonitor.java` with the following content:

```
package apt.tutorial.two;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;

public class PostMonitor extends Service {
    @Override
    public void onCreate() {
        super.onCreate();
```

```
}

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

@Override
public void onDestroy() {
    super.onDestroy();
}
}
```

We also need to add our service to `AndroidManifest.xml`, so add `<service android:name = ".PostMonitor" />` inside the `<application>` element.

At this point, your project will still compile and run, though our service is not yet doing anything.

Step #2: Set Up a Background Thread

Next, we need a well-managed background thread that can do our timeline polling for us.

First, add an import for `java.util.concurrent.atomic.AtomicBoolean`, along with a private `AtomicBoolean` data member named `active`. Initialize this to `true`.

Also, define a static int value named `POLL_PERIOD` to be the amount of time you want between timeline polling operations. Please make this no less than `60000` (one minute in milliseconds).

Then, create a `Runnable` named `threadBody` that will loop, sleeping `POLL_PERIOD` each pass, until `active` is toggled to `false`:

```
private Runnable threadBody=new Runnable() {
    public void run() {
        while (active.get()) {
            // do something with Twitter
            SystemClock.sleep(POLL_PERIOD);
        }
    }
}
```

```
    }  
};
```

In `onCreate()`, add a statement to start up our background thread on `threadBody`:

```
@Override  
public void onCreate() {  
    super.onCreate();  
  
    new Thread(threadBody).start();  
}
```

Finally, in `onDestroy()`, set the `active` flag to `false`. Once the background thread wakes up from its current sleep cycle, it will see the `false` flag and fall out of its loop, terminating its thread:

```
@Override  
public void onDestroy() {  
    super.onDestroy();  
  
    active.set(false);  
}
```

Step #3: Poll Your Friends

Now, we have a service that will do work in the background. We just need to set up the work itself.

What we want to do is load our timeline (with our status and those from our friends) every poll period and find out those that are new. To do this, we need to create a `Twitter` object, use it to load the timeline, track the already-seen status messages, and identify the new ones. Right now, let us focus on loading the timeline.

First, add an import to `winterwell.jtwitter.Twitter` so we can create `Twitter` objects. We will also need `java.util.List` in a moment, so add an import for it as well.

Then, in our `threadBody` loop, add a call to a `poll()` method:

```
private Runnable threadBody=new Runnable() {
    public void run() {
        while (active.get()) {
            poll();
            SystemClock.sleep(POLL_PERIOD);
        }
    }
};
```

Finally, add this as the implementation of `poll()` in `PostMonitor`:

```
private void poll() {
    Twitter client=new Twitter(); // need credentials!
    List<Twitter.Status> timeline=client.getFriendsTimeline();
}
```

While this will compile, it will not run, since we do not have our user name or password. We will get that from the `Patchy` client of our service in the next tutorial.

Right now, all we are doing in `poll()` is creating a fresh `Twitter` object and using it to load the timeline via `getFriendsTimeline()`.

Step #4: Find New Statuses

Once we have our timeline, we need to figure out which of the statuses are new. On the first poll, all will be new; subsequent polls should have a handful of new statuses, if any.

For the time being, let us track the already-seen status IDs via a `Set<Long>`, so add the following data member (and the `HashSet` and `Set` imports to match):

```
private Set<Long> seenStatus=new HashSet<Long>();
```

Then, we can walk the timeline, see if each status ID is in the `Set`, and process those that are not, via this change to `poll()`:

```
private void poll() {
    Twitter client=new Twitter(); // need credentials!
    List<Twitter.Status> timeline=client.getFriendsTimeline();

    for (Twitter.Status s : timeline) {
        if (!seenStatus.contains(s.id)) {
            // found a new one!
            seenStatus.add(s.id);
        }
    }
}
```

Of course, eventually, we will need to alert `Patchy` to the new statuses, but we can leave that for the next tutorial.

Step #5: Set up the Public API

Finally, we need to begin the process of adding a public API that the `Patchy` activity will be able to access.

First, we should define a Java interface that will represent our public API. Create a `src/apt/tutorial/IPostMonitor.java` file with the following content:

```
package apt.tutorial;

public interface IPostMonitor {
```

Then, we need to create a "binder" that implements this API. The binder will be supplied to `Patchy` when it binds to the `PostMonitor` service. So, add the following inner class to `PostMonitor`:

```
public class LocalBinder extends Binder implements IPostMonitor {
    void registerAccount(String user, String password) {
    }
}
```

Next, create an instance of our `LocalBinder` as part of initializing our `PostMonitor` instance:

```
private final Binder binder=new LocalBinder();
```

Finally, return that binder from `onBind()`:

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

Of course, this API does not do much, but we will expand it shortly.

At this point, you can compile your project, but it will not run properly, because we have not provided account information to the JTwitter API. That is fine for now, as we will correct that flaw in the next tutorial.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Add a preference for the timeline polling period. Use that polling period in the service, including changing the polling period when the preference changes.

Further Reading

You can learn more about the roles of services and how to create them in the "Creating a Service" chapter of [The Busy Coder's Guide to Android Development](#).

No, Really Listening To Your Friends

In the previous tutorial, we set up a service to watch for new timeline entries from our friends. The service merely finds out about these new statuses – now we need to get those someplace for our activity to display them.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 17-CreateService edition of Patchy to use as a starting point.

Step #1: Define the Callback

We need to provide some sort of callback to the service, so our activity can be notified when new status updates are available. With that in mind, create an `Patchy/src/apt/tutorial/IPostListener.java` interface as follows:

```
package apt.tutorial;

public interface IPostListener {
    void newFriendStatus(String friend, String status,
                         String createdAt);
}
```

Step #2: Enable Callbacks in the Service

Now we need to set up a means for the service client (`Patchy`) to start the service and register an `IPostListener` that will be invoked when a new status arrives. The callback will be associated with a user name and password, so we have the credentials with which to call identi.ca via its API.

First, let us create an inner class in `PostMonitor` to represent the callback and login credentials. Add this implementation of `Account` to your `PostMonitor`:

```
class Account {  
    String user=null;  
    String password=null;  
    IPostListener callback=null;  
  
    Account(String user, String password,  
            IPostListener callback) {  
        this.user=user;  
        this.password=password;  
        this.callback=callback;  
    }  
}
```

Next, import `java.util.Map` and `java.util.concurrent.ConcurrentHashMap` and set up a private data member representing the roster of outstanding accounts:

```
private Map<IPostListener, Account> accounts=  
    new ConcurrentHashMap<IPostListener, Account>();
```

Now, we can set up public APIs on our binder for our client to use to register and remove an account, by extending `IPostMonitor`:

```
package apt.tutorial;  
  
import apt.tutorial.IPostListener;  
  
public interface IPostMonitor {  
    void registerAccount(String user, String password,  
                         IPostListener callback);  
    void removeAccount(IPostListener callback);  
}
```

...and by augmenting its implementation:

```
public class LocalBinder extends Binder implements IPostMonitor {
    public void registerAccount(String user, String password,
                                IPostListener callback) {
        Account l=new Account(user, password, callback);
        poll(l);
        accounts.put(callback, l);
    }

    public void removeAccount(IPostListener callback) {
        accounts.remove(callback);
    }
}
```

Notice that we call `poll()` with our `Account` when it is registered. This will fetch our current timeline right away, rather than wait for our polling loop to cycle back around. The downside is that this `poll()` is called on the main thread rather than a background thread – we will address this in a later tutorial.

Next, update our `threadBody` to make use of our roster of `Account` objects:

```
private Runnable threadBody=new Runnable() {
    public void run() {
        while (active.get()) {
            for (Account l : accounts.values()) {
                poll(l);
            }
            SystemClock.sleep(POLL_PERIOD);
        }
    }
};
```

This too calls `poll()` with the `Account` information. Hence, we need to update `poll()` to use the `Account` and call the callback on new status messages:

```
private void poll(Account l) {
    try {
        Twitter client=new Twitter(l.user, l.password);
        client.setAPIRootUrl("https://identi.ca/api");
```

```
List<Twitter.Status> timeline=client.getFriendsTimeline();  
  
for (Twitter.Status s : timeline) {  
    if (!seenStatus.contains(s.id)) {  
        l.callback.newFriendStatus(s.user.screenName, s.text,  
            s.createdAt.toString());  
        seenStatus.add(s.id);  
    }  
}  
}  
catch (Throwable t) {  
    android.util.Log.e("PostMonitor",  
        "Exception in poll()", t);  
}  
}
```

Step #3: Manage the Service and Register the Account

Next, we need to take steps in Patchy to connect to the PostMonitor service and arrange to get callbacks when new timeline entries are ready.

First, modify `onCreate()` in Patchy to call `bindService()`, which will auto-create our PostMonitor service and give us control when it has been started and bound:

```
bindService(new Intent(this, PostMonitor.class), svcConn,  
    BIND_AUTO_CREATE);
```

Specifically, it gives us control via a `ServiceConnection` object, which you will need to add as a data member of Patchy:

```
private ServiceConnection svcConn=new ServiceConnection() {  
    public void onServiceConnected(ComponentName className,  
        IBinder binder) {  
        service=(IPostMonitor)binder;  
  
        try {  
            service.registerAccount(prefs.getString("user", null),  
                prefs.getString("password", null),  
                listener);  
        }  
        catch (Throwable t) {  
            Log.e("Patchy", "Exception in call to registerAccount()", t);  
            goBlooey(t);  
        }  
    }  
}
```

```
public void onServiceDisconnected(ComponentName className) {  
    service=null;  
}  
};
```

Notice how it will obtain our `IPostMonitor` API via the "binder" object passed to it – since this is a local service, in the same VM, this is the actual `LocalBinder` object from `PostMonitor`. Also note that we register our account information and an `IPostListener` instance at this point.

We need to unbind our service, after removing our account, when we are done, so add an implementation of `onDestroy()` to `Patchy` as follows:

```
@Override  
public void onDestroy() {  
    super.onDestroy();  
  
    service.removeAccount(listener);  
    unbindService(svcConn);  
}
```

Our code in `onCreate()` refers to a `listener` instance, so add a definition of it as an `IPostListener` implementation to `Patchy`:

```
private IPostListener listener=new IPostListener() {  
    public void newFriendStatus(String friend, String status,  
                               String createdAt) {  
    }  
};
```

Right now, this listener does not do anything – we will address that minor shortcoming in the next section.

At this point, we register our account at startup and remove it on shutdown. What we are missing is account changes. If the user, via the preferences, adjusts the user name or password, we need to get that information to the service. The cleanest way to do that is to remove our current account and register a fresh one.

With that in mind, modify `resetClient()` to handle that chore:

```
synchronized private void resetClient() {
    client=null;
    service.removeAccount(listener);
    service.registerAccount(prefs.getString("user", ""),
                           prefs.getString("password", ""),
                           listener);
}
```

Step #4: Display the Timeline

At this point, Patchy is fully integrated with the service and should receive callbacks on timeline changes. However, it does not do anything with those changes. So, we need to add something to our UI to actually display the timeline. One easy way to do that is via a `ListView` and custom row layout – that way, the timeline can be as long as we want, and it will scroll to show all of the entries.

So, modify `Patchy/res/layout/main.xml` to add a `ListView`, as shown below:

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:stretchColumns="1"
    >
    <TableRow>
        <TextView android:text="Status:" />
        <EditText android:id="@+id/status"
            android:singleLine="false"
            android:gravity="top"
            android:lines="5"
            android:scrollHorizontally="false"
            android:maxLines="5"
            android:maxLength="200sp"
        />
    </TableRow>
    <Button android:id="@+id/send"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Send"
    />
    <ListView android:id="@+id/timeline"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
    />
</TableLayout>
```

We also need to create a layout for our rows. We have three pieces of information to display: the screen name of the friend, the status message, and the date the status was modified. Create `Patchy/res/layout/row.xml` with the following layout to display all three of those pieces:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="4px"
    >
    <LinearLayout
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:layout_weight="1"
        android:gravity="center_vertical"
        >
        <TextView android:id="@+id/friend"
            android:layout_width="0px"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:gravity="left"
            android:textStyle="bold"
            android:singleLine="true"
            android:ellipsize="end"
            />
        <TextView android:id="@+id/created_at"
            android:layout_width="0px"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:gravity="right"
            android:singleLine="true"
            android:ellipsize="end"
            />
    </LinearLayout>
    <TextView android:id="@+id/status"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:gravity="center_vertical"
        android:singleLine="false"
        />
</LinearLayout>
```

Now we need to replicate a lot of the logic from `LunchList` to populate this custom list in `Patchy`.

Add a `TimelineEntry` inner class to `Patchy` to hold a single timeline entry:

```
class TimelineEntry {  
    String friend="";  
    String createdAt="";  
    String status="";  
  
    TimelineEntry(String friend, String createdAt,  
                  String status) {  
        this.friend=friend;  
        this.createdAt=createdAt;  
        this.status=status;  
    }  
}
```

Now, add an `ArrayList` of `TimelineEntry` objects to serve as the timeline itself, as a private data member of `Patchy`:

```
private List<TimelineEntry> timeline=new ArrayList<TimelineEntry>();
```

Then, add a `TimelineEntryWrapper` that will update one of our rows with the contents of a `TimelineEntry`:

```
class TimelineEntryWrapper {  
    private TextView friend=null;  
    private TextView createdAt=null;  
    private TextView status=null;  
    private View row=null;  
  
    TimelineEntryWrapper(View row) {  
        this.row=row;  
    }  
  
    void populateFrom(TimelineEntry s) {  
        getFriend().setText(s.friend);  
        getCreatedAt().setText(s.createdAt);  
        getStatus().setText(s.status);  
    }  
  
    TextView getFriend() {  
        if (friend==null) {  
            friend=(TextView)row.findViewById(R.id.friend);  
        }  
  
        return(friend);  
    }  
  
    TextView getCreatedAt() {  
        if (createdAt==null) {  
            createdAt=(TextView)row.findViewById(R.id.created_at);  
        }  
    }  
}
```

```
    return(createdAt);
}

TextView getStatus() {
    if (status==null) {
        status=(TextView)row.findViewById(R.id.status);
    }

    return(status);
}
}
```

Next, add a `TimelineAdapter` implementation that will display our timeline:

```
class TimelineAdapter extends ArrayAdapter<TimelineEntry> {
    TimelineAdapter() {
        super(Patchy.this, R.layout.row, timeline);
    }

    public View getView(int position, View convertView,
                        ViewGroup parent) {
        View row=convertView;
        TimelineEntryWrapper wrapper=null;

        if (row==null) {
            LayoutInflater inflater=getLayoutInflater();

            row=inflater.inflate(R.layout.row, parent, false);
            wrapper=new TimelineEntryWrapper(row);
            row.setTag(wrapper);
        }
        else {
            wrapper=(TimelineEntryWrapper)row.getTag();
        }

        wrapper.populateFrom(timeline.get(position));

        return(row);
    }
}
```

We need to have a `TimelineAdapter` as a private data member, so add one (initialized to `null`), then instantiate the adapter in `onCreate()`:

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

    status=(EditText)findViewById(R.id.status);
```

```
Button send=(Button)findViewById(R.id.send);
send.setOnClickListener(onSend);

prefs=PreferenceManager.getDefaultSharedPreferences(this);
prefs.registerOnSharedPreferenceChangeListener(prefListener);

bindService(new Intent(this, PostMonitor.class), svcConn,
        BIND_AUTO_CREATE);

adapter=new TimelineAdapter();
((ListView)findViewByIId(R.id.timeline)).setAdapter(adapter);
}
```

Finally, add logic to our listener object to create a `TimelineEntry` and add it to the top of the `TimelineAdapter`, so new entries are added to the head of the list:

```
private IPostListener listener=new IPostListener() {
    public void newFriendStatus(final String friend, final String status,
                                final String createdAt) {
        runOnUiThread(new Runnable() {
            public void run() {
                adapter.insert(new TimelineEntry(friend,
                                                createdAt,
                                                status),
                            0);
            }
        });
    }
};
```

With all that behind you, recompile and reinstall the application. Now, Patchy will display your timeline and should keep the timeline current as new entries roll in:

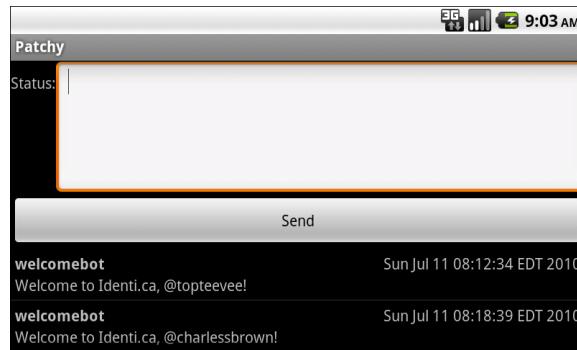


Figure 34. The timeline as displayed in Patchy

Most likely, you will find that some poll requests fail with a "TwitterException: -1" recorded in LogCat. This is due to some issue with the identi.ca Web service disconnecting some API requests. It should succeed much of the time, and so these sporadic failures are not your fault.

Here is a full implementation of Patchy after all of the above changes:

```
package apt.tutorial.two;

import android.app.Activity;
import android.app.AlertDialog;
import android.content.ComponentName;
import android.content.Intent;
import android.content.ServiceConnection;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.os.IBinder;
import android.preference.PreferenceManager;
import android.view.LayoutInflater;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.view.ViewGroup;
import android.util.Log;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ListView;
import android.widget.TextView;
import java.util.ArrayList;
import java.util.List;
import winterwell.jtwitter.Twitter;
import apt.tutorial.IPostListener;
import apt.tutorial.IPostMonitor;
```

```
public class Patchy extends Activity {
    private EditText status=null;
    private SharedPreferences prefs=null;
    private Twitter client=null;
    private List<TimelineEntry> timeline=new ArrayList<TimelineEntry>();
    private TimelineAdapter adapter=null;
    private IPostMonitor service=null;
    private ServiceConnection svcConn=new ServiceConnection() {
        public void onServiceConnected(ComponentName className,
                                       IBinder binder) {
            service=(IPostMonitor)binder;

            try {
                service.registerAccount(prefs.getString("user", null),
                                        prefs.getString("password", null),
                                        listener);
            }
            catch (Throwable t) {
                Log.e("Patchy", "Exception in call to registerAccount()", t);
                goBlooeey(t);
            }
        }

        public void onServiceDisconnected(ComponentName className) {
            service=null;
        }
    };

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

        status=(EditText)findViewById(R.id.status);

        Button send=(Button)findViewById(R.id.send);

        send.setOnClickListener(onSend);

        prefs=PreferenceManager.getDefaultSharedPreferences(this);
        prefs.registerOnSharedPreferenceChangeListener(prefListener);

        bindService(new Intent(this, PostMonitor.class), svcConn,
                   BIND_AUTO_CREATE);

        adapter=new TimelineAdapter();
        ((ListView)findViewById(R.id.timeline)).setAdapter(adapter);
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
    }
}
```

```
        service.removeAccount(listener);
        unbindService(svcConn);
    }

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    new MenuInflater(getApplicationContext())
        .inflate(R.menu.option, menu);

    return(super.onCreateOptionsMenu(menu));
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.prefs) {
        startActivity(new Intent(this, EditPreferences.class));

        return(true);
    }

    return(super.onOptionsItemSelected(item));
}

synchronized private Twitter getClient() {
    if (client==null) {
        client=new Twitter(prefs.getString("user", ""),
                            prefs.getString("password", ""));
        client.setAPIRootUrl("https://identi.ca/api");
    }

    return(client);
}

synchronized private void resetClient() {
    client=null;
    service.removeAccount(listener);
    service.registerAccount(prefs.getString("user", ""),
                           prefs.getString("password", ""),
                           listener);
}

private void updateStatus() {
    try {
        getClient().updateStatus(status.getText().toString());
    }
    catch (Throwable t) {
        Log.e("Patchy", "Exception in updateStatus()", t);
        goBlooey(t);
    }
}

private void goBlooey(Throwable t) {
    AlertDialog.Builder builder=new AlertDialog.Builder(this);
```

```
builder
    .setTitle("Exception!")
    .setMessage(t.toString())
    .setPositiveButton("OK", null)
    .show();
}

private View.OnClickListener onSend=new View.OnClickListener() {
    public void onClick(View v) {
        updateStatus();
    }
};

private SharedPreferences.OnSharedPreferenceChangeListener prefListener=
    new SharedPreferences.OnSharedPreferenceChangeListener() {
    public void onSharedPreferenceChanged(SharedPreferences sharedPrefs,
                                          String key) {
        if (key.equals("user") || key.equals("password")) {
            resetClient();
        }
    }
};

private IPostListener listener=new IPostListener() {
    public void newFriendStatus(final String friend, final String status,
                               final String createdAt) {
        runOnUiThread(new Runnable() {
            public void run() {
                adapter.insert(new TimelineEntry(friend,
                                                  createdAt,
                                                  status),
                              0);
            }
        });
    }
};

class TimelineEntry {
    String friend="";
    String createdAt="";
    String status="";

    TimelineEntry(String friend, String createdAt,
                  String status) {
        this.friend=friend;
        this.createdAt=createdAt;
        this.status=status;
    }
}

class TimelineAdapter extends ArrayAdapter<TimelineEntry> {
    TimelineAdapter() {
        super(Patchy.this, R.layout.row, timeline);
    }
}
```

```
public View getView(int position, View convertView,
    ViewGroup parent) {
    View row=convertView;
    TimelineEntryWrapper wrapper=null;

    if (row==null) {
        LayoutInflater inflater=getLayoutInflater();

        row=inflater.inflate(R.layout.row, parent, false);
        wrapper=new TimelineEntryWrapper(row);
        row.setTag(wrapper);
    }
    else {
        wrapper=(TimelineEntryWrapper)row.getTag();
    }

    wrapper.populateFrom(timeline.get(position));

    return(row);
}

class TimelineEntryWrapper {
    private TextView friend=null;
    private TextView createdAt=null;
    private TextView status=null;
    private View row=null;

    TimelineEntryWrapper(View row) {
        this.row=row;
    }

    void populateFrom(TimelineEntry s) {
        getFriend().setText(s.friend);
        getCreatedAt().setText(s.createdAt);
        getStatus().setText(s.status);
    }

    TextView getFriend() {
        if (friend==null) {
            friend=(TextView)row.findViewById(R.id.friend);
        }

        return(friend);
    }

    TextView getCreatedAt() {
        if (createdAt==null) {
            createdAt=(TextView)row.findViewById(R.id.created_at);
        }

        return(createdAt);
    }
}
```

```
    TextView getStatus() {
        if (status==null) {
            status=(TextView)row.findViewById(R.id.status);
        }

        return(status);
    }
}
```

Similarly, here is a full implementation of PostMonitor as of this point:

```
package apt.tutorial.two;

import android.app.Service;
import android.content.Intent;
import android.os.Binder;
import android.os.IBinder;
import android.os.SystemClock;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicBoolean;
import winterwell.jtwitter.Twitter;
import apt.tutorial.IPostListener;
import apt.tutorial.IPostMonitor;

public class PostMonitor extends Service {
    private static final int POLL_PERIOD=60000;
    private AtomicBoolean active=new AtomicBoolean(true);
    private Set<Long> seenStatus=new HashSet<Long>();
    private Map<IPostListener, Account> accounts=
        new ConcurrentHashMap<IPostListener, Account>();
    private final Binder binder=new LocalBinder();

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

        new Thread(threadBody).start();
    }

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

    @Override
    public void onDestroy() {
        super.onDestroy();
    }
}
```

```
    active.set(false);
}

private void poll(Account l) {
    try {
        Twitter client=new Twitter(l.user, l.password);

        client.setAPIRootUrl("https://identi.ca/api");

        List<Twitter.Status> timeline=client.getFriendsTimeline();

        for (Twitter.Status s : timeline) {
            if (!seenStatus.contains(s.id)) {
                l.callback.newFriendStatus(s.user.screenName, s.text,
                    s.createdAt.toString());
                seenStatus.add(s.id);
            }
        }
    } catch (Throwable t) {
        android.util.Log.e("PostMonitor",
            "Exception in poll()", t);
    }
}

private Runnable threadBody=new Runnable() {
    public void run() {
        while (active.get()) {
            for (Account l : accounts.values()) {
                poll(l);
            }

            SystemClock.sleep(POLL_PERIOD);
        }
    }
};

class Account {
    String user=null;
    String password=null;
    IPostListener callback=null;

    Account(String user, String password,
            IPostListener callback) {
        this.user=user;
        this.password=password;
        this.callback=callback;
    }
}

public class LocalBinder extends Binder implements IPostMonitor {
    public void registerAccount(String user, String password,
        IPostListener callback) {
```

```
Account l=new Account(user, password, callback);

    poll(1);
    accounts.put(callback, l);
}

public void removeAccount(IPostListener callback) {
    accounts.remove(callback);
}
}
```

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Switch to using a database as the storage mechanism for statuses, rather than RAM. The service can insert new statuses into the database and keep the table trimmed to some maximum number of statuses (to keep storage requirements down). The callback would notify the activity that there were updates requiring a refresh of the activity's CursorAdapter on the database. Consider using a preference to allow the user to control the maximum number of statuses to track.
- Right now, when several timeline entries are added through one API call (e.g., the initial poll), they are put in the list in the wrong order. Change Patchy and PostMonitor to have them appear in the proper order, no matter how many entries are retrieved in a single poll.
- Have the background thread running only if there is one or more registered listeners.
- LocalBinder calls poll() upon registerAccount(), to force a timeline update right away. However, this is done on the main application thread, which is a bad idea, since poll() involves network I/O. Modify LocalBinder to use an AsyncTask for the initial call to poll().

Further Reading

For more coverage of the local binding pattern and connecting to a service that way, you will want to look at the "Creating a Service" and "Invoking a Service" chapters of [The Busy Coder's Guide to Android Development](#).

Your Friends Seem Remote

In the previous example, we implemented the friends timeline via a local service. Now, let us re-implement this as a remote service, so we can allow other applications to possibly use our service.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 18-LocalService edition of Patchy to use as a starting point.

Step #1: Back Up or Branch Your Project

This tutorial is a branch off of the main flow of the tutorials. The next tutorial will have you pick up where the last tutorial left off, so that you do not need to keep dealing with the remote service throughout the entire rest of the book. Hence, you may want to back up or otherwise keep a copy of your code as it exists, so you can return to it later.

Step #2: Create a Fresh Project

First, we should create a separate project for the `PostMonitor`, to simulate it being some third-party component and to force it to run in a separate process as a separate user.

So, via `android create project` or Eclipse, create a `TMonitor` project peer of `Patchy`, using the package `apt.tutorial.three` (so `TMonitor` and `Patchy` can both be on the emulator at the same time).

Step #3: Move the Service to the New Project

Now, let us move the `PostMonitor` code from `Patchy` into the new `TMonitor` project.

First, move `Patchy/src/apt/tutorial/two/PostMonitor.java` to `TMonitor/src/apt/tutorial/three`, replacing the automatically-generated `PostMonitor.java`. Also, go in and fix up the package statement, reflecting that the new `PostMonitor` is in `apt.tutorial.three`.

Then, since both `Patchy` and `PostMonitor` need the `JTwitter` JAR, copy that jar from `Patchy/libs/` to `TMonitor/libs/`.

Next, modify `TMonitor/AndroidManifest.xml` to get rid of the `<activity>` element (since `TMonitor` has no activity) and replace it with the `<service>` element from `Patchy/AndroidManifest.xml`. While you have `Patchy/AndroidManifest.xml` open, get rid of that `<service>` element after you have put it in the `TMonitor/AndroidManifest.xml` file.

Finally, since we removed the `PostMonitor` class from `Patchy`, we need to get rid of the compiled edition of that class. If you are using Eclipse, force a rebuild of your project; otherwise, get rid of `Patchy/bin/`, so on the next recompile, it will not have any of the old code lying around.

At this point, if you recompile each project, you will see they both have compiler errors – PostMonitor cannot find IPostListener and Patchy cannot find PostMonitor, for example.

Step #4: Implement and Copy the AIDL

Now, we need to implement AIDL to represent the interface that the service exposes to the client and the callback that the client exposes to the service.

We already almost have the callback AIDL, since AIDL closely resembles Java interfaces. Move Patchy/src/apt/tutorial/two/IPostListener.java to Patchy/src/apt/tutorial/IPostListener.aidl, moving it up one directory and changing the file extension to .aidl. Then, remove the public keywords, and the AIDL is set. Repeat this process with IPostMonitor.java as well.

Then, copy the new IPostListener.aidl and IPostMonitor.aidl files to the corresponding directory in TMonitor (TMonitor/src/apt/tutorial). We also have to deal with the possibility of an android.os.RemoteException when using the callback, since the calling process might have unexpectedly terminated. So, we need to wrap our callback use in a try/catch block as follows:

```
private void poll(Account l) {
    try {
        Twitter client=new Twitter(l.user, l.password);

        client.setAPIRootUrl("https://identi.ca/api");

        List<Twitter.Status> timeline=client.getFriendsTimeline();

        for (Twitter.Status s : timeline) {
            if (!seenStatus.contains(s.id)) {
                try {
                    l.callback.newFriendStatus(s.user.screenName, s.text,
                                              s.createdAt.toString());
                seenStatus.add(s.id);
            }
            catch (Throwable t) {
                Log.e("PostMonitor", "Exception in callback", t);
            }
        }
    }
}
```

```
        }
    }
    catch (Throwable t) {
        Log.e("PostMonitor", "Exception in poll()", t);
    }
}
```

Step #5: Implement the Client Side

Next, we need to fix up the `Patchy` client, so it uses the remote service interface instead of attempting to access a local `PostMonitor`.

Then, we need to make one slight modification to our `IPostListener` implementation: our anonymous inner class actually has to implement `IPostListener.Stub` instead of `IPostListener`. So, change the listener implementation to:

```
private IPostListener listener=new IPostListener.Stub() {
    public void newFriendStatus(final String friend,
                                final String status,
                                final String createdAt) {
        runOnUiThread(new Runnable() {
            public void run() {
                adapter.insert(new TimelineEntry(friend,
                                                   createdAt,
                                                   status), 0);
            }
        });
    }
};
```

Next, modify your `ServiceConnection` data member named `svcConn` to have following implementation, to change the way we access our service interface:

```
private ServiceConnection svcConn=new ServiceConnection() {
    public void onServiceConnected(ComponentName className,
                                   IBinder binder) {
        service=IPostMonitor.Stub.asInterface(binder);

        try {
            service.registerAccount(prefs.getString("user", null),
                                   prefs.getString("password", null),
                                   listener);
        }
    }
};
```

```
        catch (Throwable t) {
            Log.e("Patchy", "Exception in call to registerAccount()", t);
            goBloody(t);
        }

    public void onServiceDisconnected(ComponentName className) {
        service=null;
    }
};
```

Then, in `onDestroy()`, wrap the `removeAccount()` in a try/catch block, to deal with the possibility that our connection to the `PostMonitor` service has been broken for some reason:

```
@Override
public void onDestroy() {
    super.onDestroy();

    try {
        service.removeAccount(listener);
    }
    catch (Throwable t) {
        Log.e("Patchy", "Exception in call to removeAccount()", t);
        goBloody(t);
    }

    unbindService(svcConn);
}
```

We also need to change `resetClient()` to wrap its service API calls in a try/catch block, as follows:

```
synchronized private void resetClient() {
    client=null;

    try {
        service.removeAccount(listener);
        service.registerAccount(prefs.getString("user", ""),
                               prefs.getString("password", ""),
                               listener);
    }
    catch (Throwable t) {
        Log.e("Patchy", "Exception in resetClient()", t);
        goBloody(t);
    }
}
```

At this point, the `Patchy` project should compile cleanly. It will not run, though, without the corresponding service.

Step #6: Implement the Service Side

Finally, we need to make similar sorts of changes on the `PostMonitor` service.

First, modify `TMonitor/AndroidManifest.xml` to add an `<intent-filter>` (so `Patchy` can reference it from another process and package) and add the `INTERNET` permission that `JTwitter` will need:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="apt.tutorial.three"
    android:versionCode="1"
    android:versionName="1.0">
    <uses-permission android:name="android.permission.INTERNET" />
    <application android:label="@string/app_name">
        <service android:name=".PostMonitor">
            <intent-filter>
                <action android:name="apt.tutorial.IPostMonitor" />
            </intent-filter>
        </service>
    </application>
</manifest>
```

Next, change our `binder` object from being an instance of `LocalBinder` to being an instance of an `IPostMonitor.Stub` anonymous inner class instance as follows:

```
private final IPostMonitor.Stub binder=new IPostMonitor.Stub() {
    public void registerAccount(String user, String password,
                                IPostListener callback) {
        Account l=new Account(user, password, callback);

        poll(1);
        accounts.put(callback, l);
    }

    public void removeAccount(IPostListener callback) {
        accounts.remove(callback);
    }
};
```

The easiest way to make the above change is to change the first line of the `LocalBinder` definition to the first line in the above listing, add the semicolon at the end, and get rid of the separate declaration and initializer of `binder`.

Then, we need to do something about our polling logic, so when we register an account we do not tie up the UI thread of the client doing an immediate `poll()`, yet we still quickly get status updates rather than waiting a full minute. To help resolve this, we will use two polling periods: a one-second period when we have no accounts, and the original one-minute period for when we do have accounts. That way, we will find out about our first account within one second of it being registered; only then will we slow down, so as not to abuse the service.

To that end, add the following data members to `PostMonitor`:

```
private static final int INITIAL_POLL_PERIOD=1000;
private int pollPeriod=INITIAL_POLL_PERIOD;
```

Also, change `threadBody` to use the new polling logic:

```
private Runnable threadBody=new Runnable() {
    public void run() {
        while (active.get()) {
            for (Account l : accounts.values()) {
                poll(l);
                pollPeriod=POLL_PERIOD;
            }
            SystemClock.sleep(pollPeriod);
        }
    }
};
```

Now, you should be able to recompile and reinstall both projects. Launching `Patchy` will give you similar results as before – the difference being that now it is using the remote `PostMonitor` service, rather than a local one.

Here is an implementation of `Patchy` after making this tutorial's modifications:

```
package apt.tutorial.two;

import android.app.Activity;
import android.app.AlertDialog;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.os.IBinder;
import android.preference.PreferenceManager;
import android.view.LayoutInflater;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.view.ViewGroup;
import android.util.Log;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ListView;
import android.widget.TextView;
import java.util.ArrayList;
import java.util.List;
import winterwell.jtwitter.Twitter;
import apt.tutorial.IPostListener;
import apt.tutorial.IPostMonitor;

public class Patchy extends Activity {
    private EditText status=null;
    private SharedPreferences prefs=null;
    private Twitter client=null;
    private List<TimelineEntry> timeline=new ArrayList<TimelineEntry>();
    private TimelineAdapter adapter=null;
    private IPostMonitor service=null;
    private ServiceConnection svcConn=new ServiceConnection() {
        public void onServiceConnected(ComponentName className,
                                      IBinder binder) {
            service=IPostMonitor.Stub.asInterface(binder);

            try {
                service.registerAccount(prefs.getString("user", null),
                                       prefs.getString("password", null),
                                       listener);
            }
            catch (Throwable t) {
                Log.e("Patchy", "Exception in call to registerAccount()", t);
                goBloody(t);
            }
        }

        public void onServiceDisconnected(ComponentName className) {
            service=null;
        }
    };
}
```

```
}

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

    status=(EditText)findViewById(R.id.status);

    Button send=(Button)findViewById(R.id.send);

    send.setOnClickListener(onSend);

    prefs=PreferenceManager.getDefaultSharedPreferences(this);
    prefs.registerOnSharedPreferenceChangeListener(prefListener);

    bindService(new Intent(IPostMonitor.class.getName()),
               svcConn, Context.BIND_AUTO_CREATE);

    adapter=new TimelineAdapter();
    ((ListView)findViewById(R.id.timeline)).setAdapter(adapter);
}

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

    try {
        service.removeAccount(listener);
    }
    catch (Throwable t) {
        Log.e("Patchy", "Exception in call to removeAccount()", t);
        goBlooey(t);
    }

    unbindService(svcConn);
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    new MenuInflater(getApplicationContext())
        .inflate(R.menu.option, menu);

    return(super.onCreateOptionsMenu(menu));
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.prefs) {
        startActivity(new Intent(this, EditPreferences.class));

        return(true);
    }
}
```

```
    return(super.onOptionsItemSelected(item));
}

synchronized private Twitter getClient() {
    if (client==null) {
        client=new Twitter(prefs.getString("user", ""),
                           prefs.getString("password", ""));
        client.setAPIRootUrl("https://identi.ca/api");
    }

    return(client);
}

synchronized private void resetClient() {
    client=null;

    try {
        service.removeAccount(listener);
        service.registerAccount(prefs.getString("user", ""),
                               prefs.getString("password", ""),
                               listener);
    }
    catch (Throwable t) {
        Log.e("Patchy", "Exception in resetClient()", t);
        goBlooeey(t);
    }
}

private void updateStatus() {
    try {
        getClient().updateStatus(status.getText().toString());
    }
    catch (Throwable t) {
        Log.e("Patchy", "Exception in updateStatus()", t);
        goBlooeey(t);
    }
}

private void goBlooeey(Throwable t) {
    AlertDialog.Builder builder=new AlertDialog.Builder(this);

    builder
        .setTitle("Exception!")
        .setMessage(t.toString())
        .setPositiveButton("OK", null)
        .show();
}

private View.OnClickListener onSend=new View.OnClickListener() {
    public void onClick(View v) {
        updateStatus();
    }
};

private SharedPreferences.OnSharedPreferenceChangeListener prefListener=
```

```
new SharedPreferences.OnSharedPreferenceChangeListener() {
    public void onSharedPreferenceChanged(SharedPreferences sharedPrefs, String key) {
        if (key.equals("user") || key.equals("password")) {
            resetClient();
        }
    }
};

private IPostListener listener=new IPostListener.Stub() {
    public void newFriendStatus(final String friend,
                               final String status,
                               final String createdAt) {
        runOnUiThread(new Runnable() {
            public void run() {
                adapter.insert(new TimelineEntry(friend,
                                                  createdAt,
                                                  status), 0);
            }
        });
    }
};

class TimelineEntry {
    String friend="";
    String createdAt="";
    String status="";

    TimelineEntry(String friend, String createdAt,
                  String status) {
        this.friend=friend;
        this.createdAt=createdAt;
        this.status=status;
    }
}

class TimelineAdapter extends ArrayAdapter<TimelineEntry> {
    TimelineAdapter() {
        super(Patchy.this, R.layout.row, timeline);
    }

    public View getView(int position, View convertView,
                       ViewGroup parent) {
        View row=convertView;
        TimelineEntryWrapper wrapper=null;

        if (row==null) {
            LayoutInflator inflater=getLayoutInflater();

            row=inflater.inflate(R.layout.row, null);
            wrapper=new TimelineEntryWrapper(row);
            row.setTag(wrapper);
        }
        else {
            wrapper=(TimelineEntryWrapper)row.getTag();
        }
    }
}
```

```
}

    wrapper.populateFrom(timeline.get(position));

    return(row);
}

class TimelineEntryWrapper {
    private TextView friend=null;
    private TextView createdAt=null;
    private TextView status=null;
    private View row=null;

    TimelineEntryWrapper(View row) {
        this.row=row;
    }

    void populateFrom(TimelineEntry s) {
        getFriend().setText(s.friend);
        getCreatedAt().setText(s.createdAt);
        getStatus().setText(s.status);
    }

    TextView getFriend() {
        if (friend==null) {
            friend=(TextView)row.findViewById(R.id.friend);
        }

        return(friend);
    }

    TextView getCreatedAt() {
        if (createdAt==null) {
            createdAt=(TextView)row.findViewById(R.id.created_at);
        }

        return(createdAt);
    }

    TextView getStatus() {
        if (status==null) {
            status=(TextView)row.findViewById(R.id.status);
        }

        return(status);
    }
}
```

And here is an implementation of PostMonitor after making this tutorial's modifications:

```
package apt.tutorial.three;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.os.SystemClock;
import android.util.Log;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicBoolean;
import winterwell.jtwitter.Twitter;
import apt.tutorial.IPostListener;
import apt.tutorial.IPostMonitor;

public class PostMonitor extends Service {
    private static final int POLL_PERIOD=60000;
    private static final int INITIAL_POLL_PERIOD=1000;
    private int pollPeriod=INITIAL_POLL_PERIOD;
    private AtomicBoolean active=new AtomicBoolean(true);
    private Set<Long> seenStatus=new HashSet<Long>();
    private Map<IPostListener, Account> accounts=
        new ConcurrentHashMap<IPostListener, Account>();

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

        new Thread(threadBody).start();
    }

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

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

        active.set(false);
    }

    private void poll(Account l) {
        try {
            Twitter client=new Twitter(l.user, l.password);

            client.setAPIRootUrl("https://identi.ca/api");

            List<Twitter.Status> timeline=client.getFriendsTimeline();

            for (Twitter.Status s : timeline) {
                if (!seenStatus.contains(s.id)) {

```

```
try {
    1.callback.newFriendStatus(s.user.screenName, s.text,
                               s.createdAt.toString());
    seenStatus.add(s.id);
}
catch (Throwable t) {
    Log.e("PostMonitor", "Exception in callback", t);
}
}
catch (Throwable t) {
    Log.e("PostMonitor", "Exception in poll()", t);
}

private Runnable threadBody=new Runnable() {
    public void run() {
        while (active.get()) {
            for (Account l : accounts.values()) {
                poll(l);
                pollPeriod=POLL_PERIOD;
            }

            SystemClock.sleep(pollPeriod);
        }
    }
};

class Account {
    String user=null;
    String password=null;
    IPostListener callback=null;

    Account(String user, String password,
            IPostListener callback) {
        this.user=user;
        this.password=password;
        this.callback=callback;
    }
}

private final IPostMonitor.Stub binder=new IPostMonitor.Stub() {
    public void registerAccount(String user, String password,
                                IPostListener callback) {
        Account l=new Account(user, password, callback);

        poll(l);
        accounts.put(callback, l);
    }

    public void removeAccount(IPostListener callback) {
        accounts.remove(callback);
    }
}
```

```
};  
}
```

Step #7: Restore Your Project

In Step #1, you should have backed up or otherwise saved a copy of your project. After doing any desired "extra credit" items, restore that project, so you are in position to continue with the rest of the tutorials. You can also get rid of the `TMonitor` application from your device or emulator, via going into the `Settings` application in the launcher, choosing `Applications`, then choosing `Manage Applications` and uninstalling `TMonitor`.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Design two other applications that could use identi.ca services (perhaps beyond those you have already implemented) and could take advantage of a common identi.ca component.

Further Reading

Learn more about remote services and AIDL in the "Your Own (Advanced) Services" chapter of [The Busy Coder's Guide to Advanced Android Development](#).

A Subtle Notification

Right now, the only way you find out about status updates from your friends is by looking at your application. However, perhaps there is a certain keyword that you want to specifically watch for – perhaps your company name. In this tutorial, we augment `Patchy` and `PostMonitor` to watch for such a keyword, putting a `Notification` in the status bar when there is a match.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the `18-LocalService` edition of `Patchy` to use as a starting point.

Step #1: Pick a Word and Icon

`PostMonitor` will watch for posts containing some word (or phrase) that you choose...but you need to choose it. Determine what you want to watch for, then add it as a `NODEFY_KEYWORD` value in `PostMonitor`:

```
private static final String NOTIFY_KEYWORD="snicklefritz";
```

You will also need an icon, named `status.png`, in your `res/drawable/` resource directory, that will be used with the `Notification`. You can grab

something suitable from the Android SDK (look for `status_*` icons), draw one yourself, or use the one supplied in the [source code for this book](#).

Step #2: Raise the Notification

Next, we need to add a method to `PostMonitor` that will use the `NotificationManager` and raise a `Notification`. Add the following `showNotification()` method to `PostMonitor`:

```
private void showNotification() {
    final NotificationManager mgr=
        (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
    Notification note=new Notification(R.drawable.status,
                                       "New matching post!",
                                       System.currentTimeMillis());
    Intent i=new Intent(this, Patchy.class);
    i.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP |
               Intent.FLAG_ACTIVITY_SINGLE_TOP);

    PendingIntent pi=PendingIntent.getActivity(this, 0,
                                              i,
                                              0);

    note.setLatestEventInfo(this, "Identica Post!",
                           "Found your keyword: "+NOTIFY_KEYWORD,
                           pi);

    mgr.notify(NOTIFICATION_ID, note);
}
```

Here we get access to the `NotificationManager` via `getSystemService()`, create an configure a `Notification` object, and tell the `NotificationManager` to show the `Notification`.

The one part of this that is a bit tricky is the work with the `Intent` and `PendingIntent`. A `PendingIntent` is a wrapper around an `Intent`, stating how it should be used (in our case, to start an `Activity`) and allowing some other process to execute this `Intent` using our security profile. In this case, this means that the operating system will call `startActivity()` on the `Intent` using the `Patchy` application's set of permissions, not the permissions of the operating system itself.

The Intent has a pair of flags added, FLAG_ACTIVITY_CLEAR_TOP and FLAG_ACTIVITY_SINGLE_TOP. The combination of these means that if there is a copy of Patchy presently active, it will be brought to the foreground, rather than start a new copy of Patchy.

To use this new method, you will also need to declare the NOTIFICATION_ID value in PostMonitor:

```
public static final int NOTIFICATION_ID=1337;
```

You will also need to add imports for android.app.Notification, android.app.NotificationManager, and android.app.PendingIntent.

Step #3: Watch for the Keyword

Armed with the implementation of showNotification(), we can now modify the poll() method to watch for hits on the keyword, so make your implementation of poll() in PostMonitor look like this:

```
private void poll(Account l) {
    try {
        Twitter client=new Twitter(l.user, l.password);

        client.setAPIRootUrl("https://identi.ca/api");

        List<Twitter.Status> timeline=client.getFriendsTimeline();

        for (Twitter.Status s : timeline) {
            if (!seenStatus.contains(s.id)) {
                l.callback.newFriendStatus(s.user.screenName, s.text,
                    s.createdAt.toString());
                seenStatus.add(s.id);

                if (s.text.indexOf(NOTIFY_KEYWORD)>-1) {
                    showNotification();
                }
            }
        }
    } catch (Throwable t) {
        android.util.Log.e("PostMonitor",
            "Exception in poll()", t);
    }
}
```

At this point, if you compile and reinstall the application, then update your own status containing your keyword, you will see the notification icon appear:

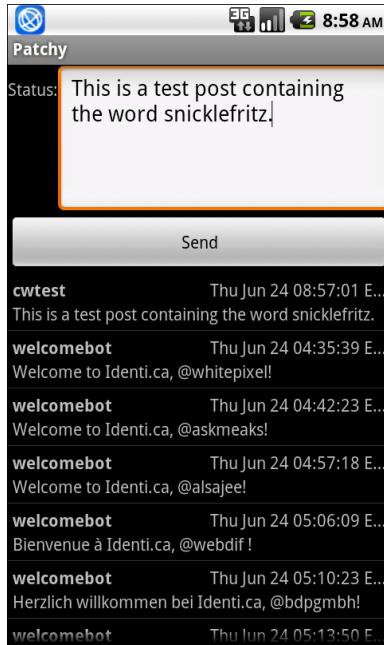


Figure 35. The Patchy notification icon

And, if you slide down the notification drawer, you will see the event information:

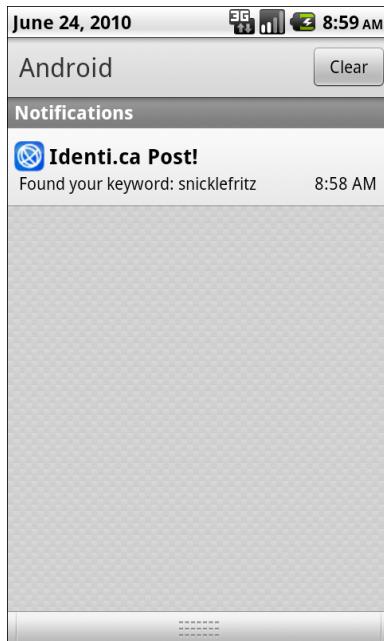


Figure 36. The Patchy notification event information

If you tap on that event, you will be returned to Patchy, even if you had closed out of it earlier. However, the icon will stick around, even after re-opening Patchy, unless you manually clear it using the Clear button in the notification drawer.

Step #4: Clearing the Notification

What would be nice is to clear the Notification automatically once the user clicks on it. To do this, first implement a `clearNotification()` method in Patchy:

```
private void clearNotification() {
    NotificationManager mgr=
        (NotificationManager) getSystemService(NOTIFICATION_SERVICE);

    mgr.cancel(PostMonitor.NOTIFICATION_ID);
}
```

Then, make a call to `clearNotification()` from `onCreate()`:

```
@Override  
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.main);  
  
    status=(EditText)findViewById(R.id.status);  
  
    Button send=(Button)findViewById(R.id.send);  
  
    send.setOnClickListener(onSend);  
  
    prefs=PreferenceManager.getDefaultSharedPreferences(this);  
    prefs.registerOnSharedPreferenceChangeListener(prefListener);  
  
    bindService(new Intent(this, PostMonitor.class), svcConn,  
        BIND_AUTO_CREATE);  
  
    adapter=new TimelineAdapter();  
    ((ListView)findViewById(R.id.timeline)).setAdapter(adapter);  
  
    clearNotification();  
}
```

If you try this, it will work...if Patchy had been closed. If, however, Patchy were open, and you click on the event, the icon would remain intact.

The reason is that if Patchy were open, the currently-running Patchy is being brought to the foreground. Since a new Patchy instance is not being created, `onCreate()` does not get called.

However, in this case, a separate callback method, `onNewIntent()`, will be called. So, override `onNewIntent()` in Patchy and call `clearNotification()` from there:

```
@Override  
public void onNewIntent(Intent i) {  
    super.onNewIntent(i);  
  
    clearNotification();  
}
```

Now, the icon should go away even if Patchy were already running when you click on the event.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- If you exit and re-open Patchy, you will notice that the timeline appears to vanish. That would occur if `PostMonitor` had not yet been destroyed, and the new `Patchy` instance reconnects to it. Modify `PostMonitor` to always send over the first timeline on the initial `poll()`.
- Support a user-configurable keyword (or set of keywords) via a Preference.
- Use the count field on a `Notification` to indicate how many matches on the keyword were found in between taps on the event information in the notification drawer.

Further Reading

Notifications are covered in the "Alerting Users Via Notifications" chapter of [The Busy Coder's Guide to Android Development](#).

Posts On Location

In this tutorial, we will integrate location tracking, such that we can optionally embed our location in our status updates following the [Twittervision](#) convention.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 20-Notifications edition of Patchy to use as a starting point.

Step #1: Get the LocationManager

First, we need to arrange to have access to Android location services through the `LocationManager`.

This requires a permission, either `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION`. Since the emulator simulates GPS, which needs `ACCESS_FINE_LOCATION`, modify `Patchy/AndroidManifest.xml` to add this permission:

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

```
    android:versionName="1.0">
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<application android:label="@string/app_name">
    <activity android:name=".Patchy"
        android:label="@string/app_name">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <activity android:name=".EditPreferences">
    </activity>
    <service android:name=".PostMonitor" />
</application>
</manifest>
```

Then, add some imports to `Patchy` for location-related classes that we will need:

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

Next, add a `LocationManager` data member named `locMgr` in `Patchy`, then initialize it in `onCreate()`:

```
locMgr=(LocationManager) getSystemService(LOCATION_SERVICE);
```

At this point, we can start using the `LocationManager` for getting the location to embed in the status update.

Step #2: Register for Location Updates

Next, we need to do something to cause Android to actually activate GPS and get fixes. Just having access to `LocationManager` is insufficient. The simplest answer is to register for location updates, even if we will use another way to get our current fix, as you will see later in this tutorial.

So, add the following statement to `onCreate()` in `Patchy`, after you have initialized `locMgr` as shown in the preceding section:

```
locMgr.requestLocationUpdates(LocationManager.GPS_PROVIDER,  
                             10000, 10000.0f,  
                             onLocationChange);
```

This refers to an `onLocationChange` object, which is a `LocationListener`. We do not truly care about the updates (though, in principle, our application could do something with them). Hence, for this tutorial's purpose, all you need is a stub implementation of `LocationListener`. Add the following to `Patchy`:

```
private LocationListener onLocationChange=new LocationListener() {  
    public void onLocationChanged(Location location) {  
        // required for interface, not used  
    }  
    public void onProviderDisabled(String provider) {  
        // required for interface, not used  
    }  
    public void onProviderEnabled(String provider) {  
        // required for interface, not used  
    }  
    public void onStatusChanged(String provider, int status,  
                               Bundle extras) {  
        // required for interface, not used  
    }  
};
```

Finally, since we registered for location updates in `onCreate()`, we should remove that request in `onDestroy()`. Modify `onDestroy()` in `Patchy` to look like this:

```
@Override  
public void onDestroy() {  
    super.onDestroy();  
  
    locMgr.removeUpdates(onLocationChange);  
    service.removeAccount(listener);  
    unbindService(svcConn);  
}
```

Step #3: Add "Insert Location" Menu

Now, we need to give the user a way to inject the current location into the status update being written. The simplest way to do that is by adding a menu item to `Patchy/res/menu/option.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/prefs"
        android:title="Settings"
        android:icon="@drawable/ic_menu_preferences"
    />
    <item android:id="@+id/location"
        android:title="Insert Location"
        android:icon="@drawable/ic_menu_compass"
    />
</menu>
```

You will also need a suitable icon, such as `ic_menu_compass.png` from the Android SDK.

If you recompile and reinstall the application, you will see the new menu option, even though it does not do anything just yet:

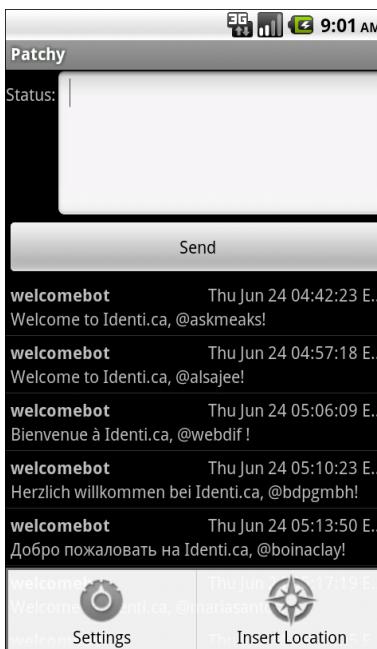


Figure 37. The Insert Location menu item in Patchy

Step #4: Insert the Last Known Location

Finally, we need to do something to add the location to our status message.

Change `onOptionsItemSelected()` in `Patchy` to look like this:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.prefs) {
        startActivity(new Intent(this, EditPreferences.class));
        return(true);
    }
    else if (item.getItemId()==R.id.location) {
        insertLocation();
        return(true);
    }
    return(super.onOptionsItemSelected(item));
}
```

Then, add in the following implementation of `insertLocation()` in `Patchy`:

```
private void insertLocation() {
    Location loc=locMgr.getLastKnownLocation(LocationManager.GPS_PROVIDER);

    if (loc==null) {
        Toast.makeText(this, "No location available", Toast.LENGTH_SHORT)
            .show();
    }
    else {
        StringBuilder buf=new StringBuilder(status
            .getText()
            .toString());
        buf.append(" L:");
        buf.append(String.valueOf(loc.getLatitude()));
        buf.append(",");
        buf.append(String.valueOf(loc.getLongitude()));
        status.setText(buf.toString());
    }
}
```

You will need to add an import for `android.widget.Toast`, used to handle the case where we do not have a GPS fix yet. Otherwise, if we have a fix, we create a piece of text in Twittervision format, using the `Location` object, and inject that into the `EditText` for the status.

Now, if you recompile and reinstall `Patchy`, and if you use DDMS to supply a fake location (assuming you are running this on an emulator), the location

will be appended to the end of the status update when you choose the Insert Location option menu item:

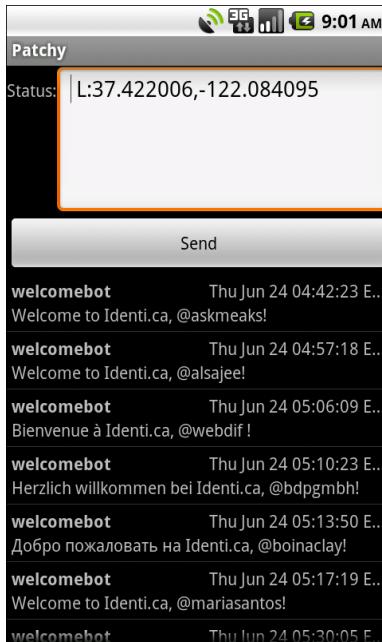


Figure 38. The results from the Insert Location menu item in Patchy

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Allow the user to choose the `LocationProvider` via a set of preferences that specify the attributes of a `Criteria` object, then using that `Criteria` object to get the best-matching `LocationProvider`.
- Use `onKeyDown()` to add "hot-key" support, such that some key (e.g., `<Alt>-<X>`) will inject the location, instead of having to use the option menu.

Further Reading

Location tracking, via GPS or other technologies, is covered in the "Accessing Location-Based Services" chapter of [The Busy Coder's Guide to Android Development](#).

Here a Post, There a Post

In this tutorial, we will integrate maps, so we can plot the location for status updates that provide such information.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 21-Location edition of Patchy to use as a starting point.

NOTE: for this application to work, you will need to have a device or emulator AVD set up with the Google APIs, and you will need to have your project set up to build with the Google APIs. If you followed the instructions in the first tutorial, this will already be done.

Step #1: Register for a Map API Key

First, you need to register for an API key to use with the mapping services and set it up in your development environment with your debug certificate. Full instructions for doing this can be found on the [Android developer site](#).

Step #2: Create a Basic MapActivity

Next, we need to create a stub `MapActivity` implementation that we can then use in Patchy.

Create `Patchy/res/layout/status_map.xml` with the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<com.google.android.maps.MapView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/map"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:apiKey="00yHj0k7_7vxbuQ9zwyXI4bNMJrAjYrJ9KKHgbQ"
    android:clickable="true" />
```

Note that you will need to substitute your API key for the one shown in `android:apiKey` in the above code listing.

Next, create `Patchy/src/apt/tutorial/two/StatusMap.java` with the following content:

```
package apt.tutorial.two;

import android.app.Activity;
import android.os.Bundle;
import android.view.ViewGroup;
import com.google.android.maps.MapActivity;
import com.google.android.maps.MapController;
import com.google.android.maps.MapView;

public class StatusMap extends MapActivity {
    private MapView map=null;

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

        map=(MapView)findViewById(R.id.map);

        map.getController().setZoom(17);
    }

    @Override
    protected boolean isRouteDisplayed() {
        return(false);
    }
}
```

```
}
```

All we do here is create a map and set the initial zoom level.

Then, we need to make two changes to our `AndroidManifest.xml` file: we need to add the `StatusMap` activity, and we need to indicate that we are using the mapping services library. Alter `Patchy/AndroidManifest.xml` to look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="apt.tutorial.two"
    android:versionCode="1"
    android:versionName="1.0">
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <application android:label="@string/app_name">
        <uses-library android:name="com.google.android.maps" />
        <activity android:name=".Patchy"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".EditPreferences">
        </activity>
        <activity android:name=".StatusMap">
        </activity>
        <service android:name=".PostMonitor" />
    </application>
</manifest>
```

Step #3: Launch the Map on Location-Bearing Status Click

Now, we need to tie `Patchy` and `StatusMap` together, such that a click on a status that has an embedded location triggers the display of the map.

To do this, first we need to get control when a status is clicked. Right now, the `onCreate()` in `Patchy` has a line that looks like:

```
((ListView) findViewById(R.id.timeline)).setAdapter(adapter);
```

Remove that line and replace it with:

```
ListView list=(ListView)findViewById(R.id.timeline);
list.setAdapter(adapter);
list.setOnItemClickListener(onStatusClick);
```

That gives control on a click to an `onStatusClick` object. In there, we need to get the `TimelineEntry` corresponding with the click, scan the status to find the location (if any), and pass that information to `StatusMap` via `startActivity()`.

Add the following `onStatusClick` implementation to `Patchy`:

```
private AdapterView.OnItemClickListener onStatusClick=
    new AdapterView.OnItemClickListener() {
    public void onItemClick(AdapterView<?> parent, View view,
        int position, long id) {
        TimelineEntry entry=timeline.get(position);
        Matcher r=regexLocation.matcher(entry.status);

        if (r.find()) {
            double latitude=Double.valueOf(r.group(1));
            double longitude=Double.valueOf(r.group(4));

            Intent i=new Intent(Patchy.this, StatusMap.class);

            i.putExtra(LATITUDE, latitude);
            i.putExtra(LONGITUDE, longitude);
            i.putExtra(STATUS_TEXT, entry.status);

            startActivity(i);
        }
    }
};
```

This requires a few constants to be added to `Patchy` as well:

```
public static final String LATITUDE="apt.tutorial.latitude";
public static final String LONGITUDE="apt.tutorial.longitude";
public static final String STATUS_TEXT="apt.tutorial.statusText";
```

We also need the Pattern that defines the regular expression we are searching for:

```
private Pattern regexLocation=Pattern.compile("L\\:\\((\\-)?[0-9]+(\\.\\.[0-9]+)?)\\,,  
((\\-)?[0-9]+(\\.\\.[0-9]+)?)" );
```

That regular expression is ugly, but it effectively matches any string of the form L:AAA,000, where AAA and 000 are floating-point numeric values.

You will also need to add imports for android.widget.AdapterView, java.util.regex.Matcher, and java.util.regex.Pattern.

Now, when we click on a location-bearing status, StatusMap will open – you can test this via the post you may have added from the previous tutorial, containing your location. However, we need to add some logic to StatusMap to unpack the latitude and longitude and center the map on that position.

Change StatusMap to look like the following:

```
package apt.tutorial.two;  
  
import android.app.Activity;  
import android.os.Bundle;  
import android.view.ViewGroup;  
import com.google.android.maps.GeoPoint;  
import com.google.android.maps.MapActivity;  
import com.google.android.maps.MapController;  
import com.google.android.maps.MapView;  
  
public class StatusMap extends MapActivity {  
    private MapView map=null;  
  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.status_map);  
  
        map=(MapView)findViewById(R.id.map);  
  
        map.getController().setZoom(17);  
  
        double lat=getIntent().getDoubleExtra(Patchy.LATITUDE, 0);  
        double lon=getIntent().getDoubleExtra(Patchy.LONGITUDE, 0);  
  
        GeoPoint status=new GeoPoint((int)(lat*1000000.0),  
                                     (int)(lon*1000000.0));  
  
        map.getController().setCenter(status);  
        map.setBuiltInZoomControls(true);  
    }  
}
```

```
@Override  
protected boolean isRouteDisplayed() {  
    return(false);  
}
```

Now, if you rebuild and reinstall Patchy (and PostMonitor!), and you click on a status that has a location (L:nnn.nn,nnn.nn – you may need somebody to update their status with a location for you!), it will open the map on that location.

Step #4: Show the Location Via a Pin

Finally, it would be good to put a pin on the map at the spot identified by the status. Not only will this help the user find the location again after panning around the map, but we can also have the pin respond to a click by displaying the text of the status update.

First, find yourself a suitable pin image, probably on the order of 32x32 pixels. In this code, we assume that the pin image is called `marker`.

Next, add the following lines to `onCreate()` in `StatusMap`:

```
String statusText=getIntent().getStringExtra(Patchy.STATUS_TEXT);  
Drawable marker=getResources().getDrawable(R.drawable.marker);  
  
marker.setBounds(0, 0, marker.getIntrinsicWidth(),  
                marker.getIntrinsicHeight());  
  
map.getOverlays().add(new StatusOverlay(marker, status,  
                                         statusText));
```

This requires an implementation of `StatusOverlay` – add the following inner class to `StatusMap`:

```
private class StatusOverlay extends ItemizedOverlay<OverlayItem> {  
    private OverlayItem item=null;  
    private Drawable marker=null;  
  
    public StatusOverlay(Drawable marker, GeoPoint status,  
                         String statusText) {
```

```
super(marker);
this.marker=marker;

item=new OverlayItem(status, "Tweet!", statusText);

populate();
}

@Override
protected OverlayItem createItem(int i) {
    return(item);
}

@Override
public void draw(Canvas canvas, MapView mapView,
                  boolean shadow) {
    super.draw(canvas, mapView, shadow);

    boundCenterBottom(marker);
}

@Override
protected boolean onTap(int i) {
    Toast.makeText(StatusMap.this,
                  item.getSnippet(),
                  Toast.LENGTH_SHORT).show();

    return(true);
}

@Override
public int size() {
    return(1);
}
```

Now, when you view the map, you will see the pin at the spot of the location:



Figure 39. The StatusMap, showing a location in New York City

Clicking on the pin will display a `Toast` with the text of the status update itself:

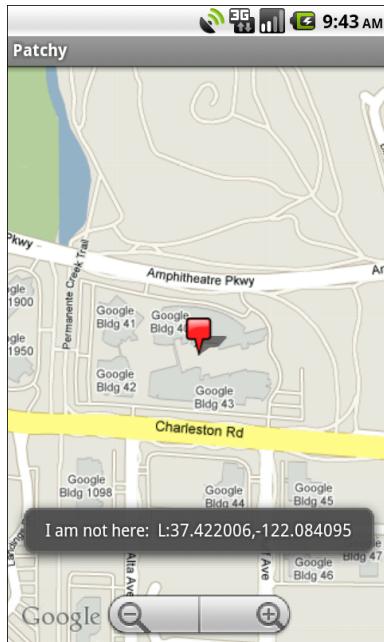


Figure 40. The StatusMap, showing a Toast of the status update associated with the location

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Support other map perspectives, such as street view or terrain mode.
- Allow the user to control the initial zoom level via a preference.
- Enable the compass rose, via `MyLocationOverlay`.

Further Reading

Integration with Google Maps is covered in the "Mapping with MapView and MapActivity" chapter of [The Busy Coder's Guide to Android Development](#).

Also, bear in mind that the documentation for Android's mapping code is not found in the Android developer guide directly, but rather at the site for the [Google add-on for Android](#).

In this tutorial, we will integrate a video clip, to serve as a placeholder for some sort of future "helpcast" to assist users with Patchy.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 22-Maps edition of Patchy to use as a starting point.

Step #1: Obtain and Install a Video Clip

First, we need a suitable video clip to serve as our helpcast placeholder. A 320x240 MP4 video file should do nicely. Please, though, respect creators' copyrights – consider using a Creative Commons-licensed clip, such as [one of these](#).

For something the size of your typical video clip, you should store it in an SD card, since space in on-board flash memory is at a premium. We need to upload the MP4 file there as `helpcast.mp4`. To do that, either use Eclipse's file-browsing tools, or use DDMS, or use the `adb push` command from the command line. The SD card can be found at either `/sdcard` or `/mnt/sdcard`, depending on your emulator or device.

Step #2: Create the Stub Helpcast Activity

Now, we need to add an activity that will use the `VideoView` widget to display our video clip.

First, add the following layout as `Patchy/res/layout/helpcast.xml`:

```
<?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"
    >
    <VideoView
        android:id="@+id/video"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
    />
</LinearLayout>
```

Then, create `Patchy/src/apt/tutorial/two/HelpCast.java` with the following code:

```
package apt.tutorial.two;

import android.app.Activity;
import android.graphics.PixelFormat;
import android.os.Bundle;
import android.os.Environment;
import android.view.View;
import android.widget.MediaController;
import android.widget.VideoView;
import java.io.File;

public class HelpCast extends Activity {
    private VideoView video;
    private MediaController ctr;

    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        getWindow().setFormat(PixelFormat.TRANSLUCENT);
        setContentView(R.layout.helpcast);

        File clip=new File(Environment.getExternalStorageDirectory(),
            "helpcast.mp4");

        if (clip.exists()) {
            video=(VideoView)findViewById(R.id.video);
```

```
        video.setVideoPath(clip.getAbsolutePath());  
  
        ctrlr=new MediaController(this);  
        ctrlr.setMediaPlayer(video);  
        video.setMediaController(ctrlr);  
        video.requestFocus();  
        video.start();  
    }  
}  
}
```

Here, we simply configure the `VideoView`, including the `MediaController` to give us buttons to control playback of the video. At the end, we call `start()`, to begin playback automatically.

And, as always, when we add an activity to our project, we need to add it to `AndroidManifest.xml`, so add the following `<activity>` element alongside the others:

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

Step #3: Launch the Helpcast from the Menu

Finally, we need to integrate `HelpCast` into `Patchy` overall, so users can display the help video. As we have done in previous tutorials, we will accomplish this by extending the option menu with another menu item.

First, add the following `<item>` to `Patchy/res/menu/options.xml`:

```
<item android:id="@+id/help"  
      android:title="Help"  
      android:icon="@drawable/ic_menu_help"  
/>
```

You will also need a suitable menu icon, such as the `ic_menu_help.png` image from the Android SDK.

Then, update `onOptionsItemSelected()` in `Patchy` to launch `HelpCast` when our item is selected:

```

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.prefs) {
        startActivity(new Intent(this, EditPreferences.class));
        return(true);
    }
    else if (item.getItemId()==R.id.location) {
        insertLocation();
        return(true);
    }
    else if (item.getItemId()==R.id.help) {
        startActivity(new Intent(this, HelpCast.class));
        return(true);
    }
    return(super.onOptionsItemSelected(item));
}

```

With all that complete, we have our helpcast ready to go. Rebuild and reinstall the application, and you will see the new help menu item:

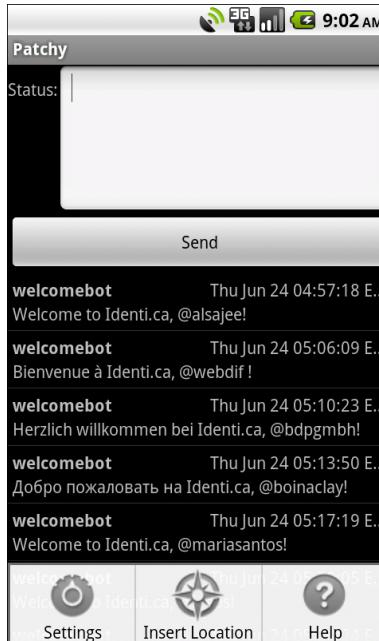


Figure 41. The new option menu item in Patchy

Clicking on it will bring up our "helpcast" video clip, though you will need to tap on the top portion of the screen to get the media controls to appear, so you can play it back:

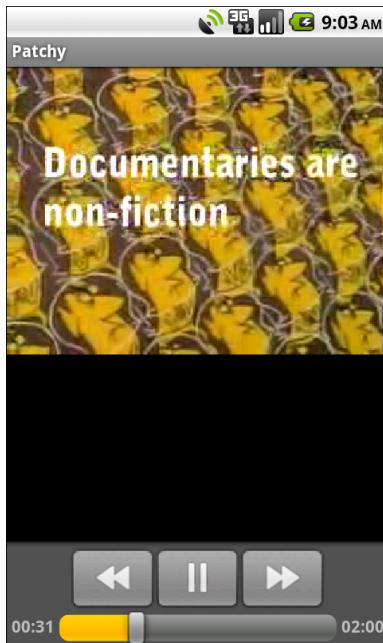


Figure 42. The "helpcast" placeholder

Note, however, that playback may or may not work, depending on the version of the emulator you are using, the file you are using, and the speed of the development machine hosting the emulator instance.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Experiment with streaming video. Note that the requirements for Android streaming video are rather arcane and ill-documented.
- If you can find several media clips that work, put a `Spinner` on the `HelpcastActivity` to allow the user to switch between video clips.

Further Reading

Instruction on how to integrate media – audio and video, local and streaming – to your Android app can be found in the "Media" chapter of [The Busy Coder's Guide to Advanced Android Development](#).

Browsing Some Posts

In this tutorial, we will support clicking on links found in status updates, popping those up in the Browser application. We will also add a help screen that loads a local help HTML file, by integrating WebKit into Patchy.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 23-Media edition of Patchy to use as a starting point.

Step #1: Add Auto-Linking

We can start by letting Android "do the heavy lifting for us" in making links in status updates clickable. That is merely a matter of adding `android:autoLink = "all"` to the `TextView` for the status text in `Patchy/res/layout/row.xml`. With a value of "all", we are saying that we want all recognized patterns to be converted to links: Web URLs, email addresses, phone numbers, etc.

If you make this change, then recompile and reinstall the application, you will notice two things:

1. All of the click-able items show up in blue underlined text, and when clicked they pop up the appropriate application (e.g., clicking a Web URL brings up the Browser application).
2. Clicking on a row with a location, as we introduced in a previous tutorial, does not spawn our StatusMap.

Step #2: Draft and Package the Help HTML

Next, we need some placeholder HTML to serve as our help prose. This does not need to be terribly fancy – in fact, simpler HTML works better, because it loads faster.

So, write a Web page that will serve as the placeholder for the `Patchy` help. The key is where you put the page: create an `assets/` directory in your project and store it as `help.html` in there. That will line up with the URL we will use in the next section to reference that help file.

Step #3: Create a Help Activity

Now, we can create a help activity class that will load our Web page and do some other useful things.

First, create `Patchy/res/layout/help.xml` with the following content:

```
<?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"
    >
    <WebView android:id="@+id/webkit"
        android:layout_width="fill_parent"
        android:layout_height="0px"
        android:layout_weight="1"
    />
</LinearLayout>
```

Then, create `Patchy/src/apt/tutorial/two/HelpPage.java` with the following code:

```
package apt.tutorial.two;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.webkit.WebView;

public class HelpPage extends Activity {
    private WebView browser;

    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.help);

        browser=(WebView)findViewById(R.id.webkit);
        browser.loadUrl("file:///android_asset/help.html");
    }
}
```

Note how we use `file:///android_asset/help.html` as the URL syntax to reach into our APK's assets to load the desired Web page.

Finally, as normal, we need to add another `<activity>` element to our `AndroidManifest.xml` file:

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

Step #4: Splice In the Help Activity

Right now, the option menu in `Patchy` for help launches the `HelpCast` activity. We want to change that so `Patchy` launches `HelpPage` instead, then augment `HelpPage` to display `HelpCast` on demand.

Making the change to display `HelpPage` is easy – just replace the `HelpCast` reference with `HelpPage` in `onOptionsItemSelected()`:

```
startActivity(new Intent(this, HelpPage.class));
```

At this point, if you rebuild and reinstall the application, then click on the help menu item, you will see your help page in a full-screen browser.

To get to HelpCast, let us add a Button to the HelpPage layout and wire it up so, when clicked, the Button launches HelpCast.

First, add a suitable Button to Patchy/res/layout/help.xml:

```
<?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"
    >
    <WebView android:id="@+id/webkit"
        android:layout_width="fill_parent"
        android:layout_height="0px"
        android:layout_weight="1"
    />
    <Button android:id="@+id/helpcast"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="View Helpcast"
    />
</LinearLayout>
```

Then, add logic to HelpPage to find the Button and set the on-click listener to an object that will call `startActivity()` on HelpCast:

```
package apt.tutorial.two;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.webkit.WebView;
import android.widget.Button;

public class HelpPage extends Activity {
    private WebView browser;

    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.help);

        browser=(WebView)findViewById(R.id.webkit);
        browser.loadUrl("file:///android_asset/help.html");

        Button cast=(Button)findViewById(R.id.helpcast);

        cast.setOnClickListener(onCast);
    }
}
```

```
private View.OnClickListener onCast=new View.OnClickListener() {
    public void onClick(View v) {
        startActivity(new Intent(HelpPage.this, HelpCast.class));
    }
};
```

Now, if you recompile and reinstall the application, clicking the help menu item brings up `HelpPage` with both help content and the "View HelpCast" button:

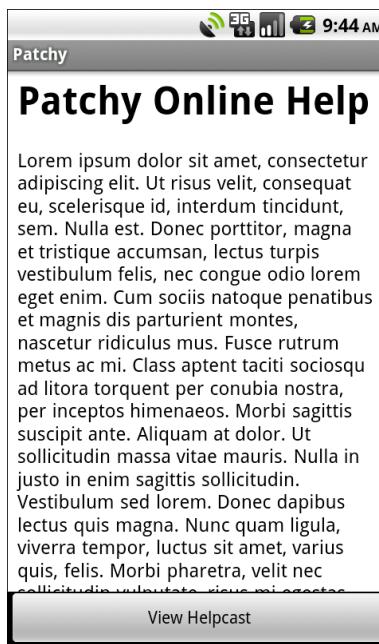


Figure 43. The HelpPage activity

Clicking the button, in turn, will display the "helpcast" video clip.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Support multiple pages of help text, by using `WebViewClient` and `shouldOverrideUrlLoading()`.

- Experiment with adding images or CSS stylesheets to the help page.
- Support the notion of the help being updated separately from the APK. Search some URL for help updates, downloading them in the background, and using them if available, falling back to the in-APK edition if there are no such updates.
- Instead of detecting locations when statuses are clicked upon, detect locations in the constructor for `TimelineEntry`. Then, add a "Map" button to the row layout, set to `GONE` if there is no location, or `VISIBLE` if there is one. Then, arrange to have clicking the Map button bring up the `StatusMap`.

Further Reading

You can learn more about the basics of integrating a `WebView` widget into your activities in the "Embedding the WebKit Browser" chapter of [The Busy Coder's Guide to Android Development](#).

High-Priced Help

In this tutorial, we will extend our help system to embed user details, such as their identi.ca screen name, in the help text itself on the fly, by way of injecting Java objects into the `WebView` Javascript engine.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 24-WebKit edition of Patchy to use as a starting point.

Step #1: Enable Javascript

By default, `WebView` widgets have Javascript disabled. Since our current help page does not use Javascript, that was not a problem in the previous tutorial. Now, however, we are looking to add a script to the page, so we need to enable Javascript in our `WebView`.

Add this line to `onCreate()` in `HelpPage`:

```
browser.getSettings().setJavaScriptEnabled(true);
```

That is all you need to enable basic Javascript operation.

Step #2: Create the Java Object to Inject

Next, we need to create a Java object from a public class with a public method that we can inject into the Javascript environment of our `WebView` that will give us access to the user's `idenit.ca` screen name.

To that end, add the following inner class to `HelpPage`:

```
public class CustomHelp {  
    SharedPreferences prefs=null;  
  
    CustomHelp() {  
        prefs=PreferenceManager  
            .getDefaultSharedPreferences(HelpPage.this);  
    }  
  
    public String getUserName() {  
        return(prefs.getString("user", "<no account>"));  
    }  
}
```

Step #3: Inject the Java Object

Just because we created the `CustomHelp` inner class does not mean Javascript has access to it. Instead, we need to give the `WebView` an instance of `CustomHelp` and associate it with a name that will be used to make the `CustomHelp` instance look like a global variable in Javascript.

Add the following lines to `onCreate()` in `HelpPage`:

```
browser.addJavascriptInterface(new CustomHelp(),  
    "customHelp");
```

The entirety of `onCreate()` in `HelpPage` should now look like this:

```
@Override  
public void onCreate(Bundle icicle) {  
    super.onCreate(icicle);  
    setContentView(R.layout.help);  
  
    browser=(WebView)findViewById(R.id.webkit);  
    browser.getSettings().setJavaScriptEnabled(true);  
    browser.addJavascriptInterface(new CustomHelp(),
```

```
        "customHelp");
browser.loadUrl("file:///android_asset/help.html");

Button cast=(Button)findViewById(R.id.helpcast);
cast.setOnClickListener(onCast);
}
```

In particular, we need to enable Javascript in the `WebView` and inject our `CustomHelp` object (under the name `customHelp`) before we load our help Web page.

Step #4: Leverage the Java Object from Javascript

With all that done, we can take advantage of the `customHelp` object in our Web page.

Somewhere on your page, add a `<div>` or `` with an id of `userName`, such as:

```
<p>Your Twitter account name is:
<span id="userName"><i>unknown</i></span>!</p>
```

Then, add a global Javascript function that will replace the default contents of the `userName` element with the value obtained from `customHelp`, such as:

```
<script language="javascript">
    function updateUserName() {
        document
            .getElementById("userName")
            .innerHTML=customHelp.getUserName();
    }
</script>
```

Finally, add an `onLoad` attribute to your `<body>` element to trigger calling the global Javascript function, such as:

```
<body onLoad="updateUserName()">
```

If you do all of that, then rebuild and reinstall the application, the `Patchy` help page should show your identi.ca user name where you placed it on the page.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Embed a link in the help HTML that will pop open the Browser application on the user's own identi.ca page
- Have the help page optionally show the user's current location as an example of what would be injected into a status message. On a location update, call a global Javascript function in the page, so it can update the help to match the current location.

Further Reading

Connecting Java and Javascript in a `WebView` widget is part of what is covered in the "WebView, Inside and Out" chapter of [The Busy Coder's Guide to Advanced Android Development](#).

PART III – Advanced Tutorials

Now Your Friends Seem Animated

Most of the time, we are reading status updates, not updating our own status. Hence, having the status entry widgets always around takes up a lot of screen space. It would be nice to have them appear or disappear at the user's request.

This tutorial will cover that very process. We will give the user an option menu choice to show or hide the status entry widgets. Initially, we will simply hide and show the widgets without any animation. Then, we will use an `AlphaAnimation` to have the widgets fade in or out.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 25-AdvWebKit edition of Patchy to use as a starting point.

Step #1: Set Up the Option Menu

Let us start by giving the user the option to show and hide the status entry row. That can be handled by just another entry in our option

menu...though we will need to do a little work to toggle the menu item from "show" to "hide" mode and back again.

First, modify `Patchy/res/layout/main.xml` to replace our `TableLayout` with some `LinearLayout` containers:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    >
    <LinearLayout android:id="@+id/status_row"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        >
        <TextView android:text="Status:"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
        />
        <EditText android:id="@+id/status"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:singleLine="false"
            android:gravity="top"
            android:lines="5"
            android:scrollHorizontally="false"
            android:maxLines="5"
            android:maxWidth="200sp"
        />
        <Button android:id="@+id/send"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="Send"
        />
    </LinearLayout>
    <ListView android:id="@+id/timeline"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        >
    />
</LinearLayout>
```

Partially, we did this because we really were not using much of the `TableLayout` functionality anymore. More importantly, though, we need to isolate our collection of widgets we want to animate into a single container – in this case, the innermost nested `LinearLayout`.

Next, add a `statusRow` View data member to `Patchy`:

```
private View statusRow=null;
```

...and initialize it somewhere late in `onCreate()`:

```
statusRow=findViewById(R.id.status_row);
```

Then, add a `status_entry` menu option to `Patchy/res/menu/option.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/prefs"
        android:title="Settings"
        android:icon="@drawable/ic_menu_preferences"
    />
    <item android:id="@+id/status_entry"
        android:title="Hide Status Entry"
        android:icon="@drawable/status_hide"
    />
    <item android:id="@+id/location"
        android:title="Insert Location"
        android:icon="@drawable/ic_menu_compass"
    />
    <item android:id="@+id/help"
        android:title="Help"
        android:icon="@drawable/ic_menu_help"
    />
</menu>
```

You will need to supply some PNG image to serve as the menu icon, preferably 32px tall.

The menu option is initially set up in "hide" mode. However, when the user pops up the menu, we want to change that menu option to "show" mode if the `StatusEntryView` is already hidden. To do this, we can use `onPrepareOptionsMenu()` – add the following method to `Patchy`:

```
@Override
public boolean onPrepareOptionsMenu(Menu menu) {
    MenuItem statusItem=menu.findItem(R.id.status_entry);

    if (statusRow.getVisibility()==View.VISIBLE) {
        statusItem.setIcon(R.drawable.status_hide);
        statusItem.setTitle("Hide Status Entry");
    }
    else {
        statusItem.setIcon(R.drawable.status_show);
    }
}
```

```
    statusItem.setTitle("Show Status Entry");
}

return(super.onPrepareOptionsMenu(menu));
}
```

Here, we see if the status entry row is visible and set our menu item accordingly.

At this point, if you recompile and reinstall Patchy, you will see the new menu item, though it will not do anything just yet:

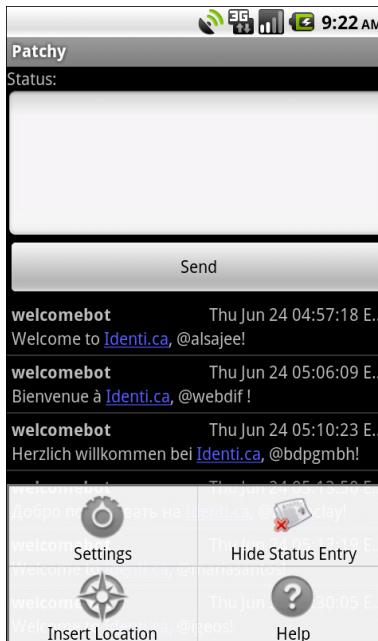


Figure 44. The "Hide Status Entry" menu option in Patchy

Step #2: Show and Hide the Status Entry Widgets

It would be nice, of course, if the newly-created menu item actually did something.

So, we need to add another case to `onOptionsItemSelected()` in Patchy to handle this new menu item:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.prefs) {
        startActivity(new Intent(this, EditPreferences.class));
        return(true);
    }
    else if (item.getItemId()==R.id.location) {
        insertLocation();
        return(true);
    }
    else if (item.getItemId()==R.id.help) {
        startActivity(new Intent(this, HelpPage.class));
        return(true);
    }
    else if (item.getItemId()==R.id.status_entry) {
        toggleStatusEntry();
        return(true);
    }
    return(super.onOptionsItemSelected(item));
}
```

Here, we merely delegate to a `toggleStatusEntry()` method, so add that method to `Patchy` as shown below:

```
private void toggleStatusEntry() {
    if (status.getVisibility()==View.VISIBLE) {
        status.setVisibility(View.GONE);
    }
    else {
        status.setVisibility(View.VISIBLE);
    }
}
```

All we do here is make the status entry row hidden (`GONE`) or visible (`VISIBLE`), depending on its current state.

Now, when you hide the status entry row, the timeline gets more room:

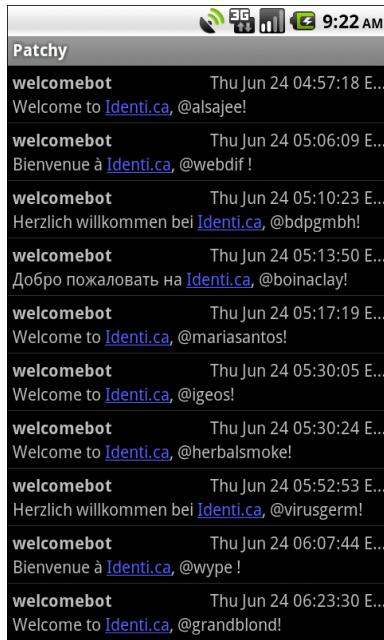


Figure 45. Patchy with a hidden status entry

Step #3: Fading In and Out

Of course, we have not yet used an animation, which is the point of this exercise.

First, we need to define our animations. In this case, we will use ones declared in animation XML files.

Create a Patchy/res/anim directory, then create Patchy/res/anim/fade_out.xml as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
    android:fromAlpha="1.0"
    android:toAlpha="0.0"
    android:duration="500" />
```

Also create Patchy/res/anim/fade_in.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
    android:fromAlpha="0.0"
    android:toAlpha="1.0"
    android:duration="500"/>
```

These set up 500-millisecond alpha animations that fade out and fade in, respectively.

Next, we need to declare Animation objects as data members in `Patchy`, to hold these once we load them:

```
private Animation fadeOut=null;
private Animation fadeIn=null;
```

Then, we can use `AnimationUtils` to load our animation XML resources into the Animation objects. Add the following statements sometime late in the `onCreate()` method in `Patchy`:

```
fadeOut=AnimationUtils.loadAnimation(this, R.anim.fade_out);
fadeOut.setAnimationListener(fadeOutListener);
fadeIn=AnimationUtils.loadAnimation(this, R.anim.fade_in);
```

You will note that in addition to loading the animation resources, we attached an `AnimationListener` to the `fadeOut` Animation. That way, we get notified when the animation is done, so we can make the widget fully "gone". Add the following implementation of `fadeOutListener` to `Patchy`:

```
Animation.AnimationListener fadeOutListener=new Animation.AnimationListener() {
    public void onAnimationEnd(Animation animation) {
        statusRow.setVisibility(View.GONE);
    }
    public void onAnimationRepeat(Animation animation) {
        // not needed
    }
    public void onAnimationStart(Animation animation) {
        // not needed
    }
};
```

Now, all that remains is to use `fadeOut` and `fadeIn` in our `toggleStatusEntry()` method:

```
private void toggleStatusEntry() {
    if (statusRow.getVisibility()==View.VISIBLE) {
        statusRow.startAnimation(fadeOut);
    }
    else {
        statusRow.setVisibility(View.VISIBLE);
        statusRow.startAnimation(fadeIn);
    }
}
```

You will also need to add imports for `android.view.animation.Animation` and `android.view.animation.AnimationUtils`.

At this point, you can rebuild and reinstall `Patchy`, and you will see your status entry area fade out when you hide it, then fade back in when you show it again.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Have the status entry widget slide out rather than fade out, either sliding out to the top or sliding out to one side.
- Use an `AnimationSet` to have the widget both fade and slide.

Further Reading

So-called "tweened" animations, such as the `AlphaAnimation` used in this chapter, are covered in the "Animating Widgets" chapter of [The Busy Coder's Guide to Advanced Android Development](#). This includes coverage of other types of tweened animations (slides, spins, etc.), how to apply several animations in parallel or sequence, etc.

Messages From The Great Beyond

In this tutorial, we will replace the callback system we used in previous editions of `Patchy` with a broadcast `Intent`. This allows `PostMonitor` to alert applications about new status updates without there having to be an explicit connection between them.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 26-Animation edition of `Patchy` to use as a starting point.

Step #1: Broadcast the Intent

Sending a broadcast `Intent` is rather simple, particularly for an application like `PostMonitor` that already collects the information to broadcast (in this case, friend, status, and creation date).

First, we need to choose a name for the action of the broadcast `Intent`. We need something that will not collide with other such names, so it is best to use something that has our namespace in it. Similarly, we need to have

names for any values we are going to store as "extras" in our Intent, and ideally those are "namespaced" as well.

With that in mind, add the following data members to PostMonitor:

```
public static final String STATUS_UPDATE="apt.tutorial.three.STATUS_UPDATE";
public static final String FRIEND="apt.tutorial.three.FRIEND";
public static final String STATUS="apt.tutorial.three.STATUS";
public static final String CREATED_AT="apt.tutorial.three.CREATED_AT";
```

Then, we need to actually broadcast the Intent, instead of calling the callback. To do that, replace your existing poll() implementation with the following:

```
private void poll(Account l) {
    try {
        Twitter client=new Twitter(l.user, l.password);

        client.setAPIRootUrl("https://identi.ca/api");

        List<Twitter.Status> timeline=client.getFriendsTimeline();

        for (Twitter.Status s : timeline) {
            if (!seenStatus.contains(s.id)) {
                try {
                    Intent broadcast=new Intent(STATUS_UPDATE);
                    broadcast.putExtra(FRIEND, s.user.screenName);
                    broadcast.putExtra(STATUS, s.text);
                    broadcast.putExtra(CREATED_AT,
                        s.createdAt.toString());
                    sendBroadcast(broadcast);
                }
                catch (Throwable t) {
                    Log.e("PostMonitor", "Exception in callback", t);
                }

                seenStatus.add(s.id);

                if (s.text.indexOf(NOTIFY_KEYWORD)>-1) {
                    showNotification();
                }
            }
        }
    }
    catch (Throwable t) {
        android.util.Log.e("PostMonitor",
            "Exception in poll()", t);
    }
}
```

We create a fresh Intent on our defined action, populate our variable data as "extras", and call sendBroadcast() to send it off.

Step #2: Catch and Use the Intent

Receiving a broadcast Intent is similarly simple.

First, we need to decide when we want our receiver to be active and alive. In some situations, where the broadcasts are merely advisory, we can enable them in onResume() and disable them in onPause(), so we consume no CPU time processing broadcasts when our activity is not visible. In this case, so we do not miss an all-important tweet, we should ensure the receiver is active whenever our account registered with PostMonitor is active.

First, we need to register a receiver in onCreate() before we bind to our service:

```
registerReceiver(receiver, new IntentFilter(PostMonitor.STATUS_UPDATE));
```

And, we need to unregister said receiver after we unbind from the service:

```
unregisterReceiver(receiver);
```

Of course, none of this will work without a receiver itself. We want the receiver to do what our callback object does, except that instead of getting the data as method parameters, we get it as "extras" on the broadcast Intent. Here is one implementation of receiver that will accomplish this end:

```
private BroadcastReceiver receiver=new BroadcastReceiver() {
    public void onReceive(Context context,
                          final Intent intent) {
        String friend=intent.getStringExtra(PostMonitor.FRIEND);
        String createdAt=intent.getStringExtra(PostMonitor.CREATED_AT);
        String status=intent.getStringExtra(PostMonitor.STATUS);

        adapter.insert(new TimelineEntry(friend, createdAt, status),
                      0);
    }
};
```

If we intend to keep support for both the callback and the broadcast Intent, we might consider refactoring our code, such that the `TimelineEntry` addition is centrally defined.

You will need to add imports for `android.content.BroadcastReceiver`, `android.content.Context`, and `android.content.IntentFilter`.

At this point, recompile and reinstall Patchy. Patchy should run as it did before, just via broadcast Intent objects instead of the callback.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Get rid of the callback API. Since the service tracks accounts by the callback object (it is the key to the `Map`), you will need to restructure how the service tracks those accounts, such as using the screen name.
- Create a separate "sniffer" project that listens for the same broadcast Intent. Confirm that it too receives the broadcasts from `PostMonitor`.

Further Reading

The use of Intent filters for broadcast Intent objects is covered briefly in the "Creating Intent Filters" chapter of [The Busy Coder's Guide to Android Development](#).

Contacting Our Friends

In this tutorial, we tie Android contacts to Patchy, storing identi.ca screen names in the contacts engine and highlighting those status updates that come from our contacts.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 27-IntentFilters edition of Patchy to use as a starting point.

Also note that this tutorial will work with an Android 2.0 device or emulator but will *not* work with an Android 1.6 or older environment. Furthermore, the Android 2.2 emulator presently seems to have a bug, whereby if you add a contact, it does not appear on the list of contacts, even though the contact is actually there.

Step #1: Fake the Contact Data

Android's contact database does not have a built-in spot to record the Twitter screen name of the contact.

Hopefully, you have gotten over the horror of that news. Fortunately, Android 1.5 added the ability to track IM addresses, and so we will use that for our contacts' identi.ca screen names.

To do this, pick a contact, or create a new one if you do not have any. Fill in the person's display name, then scroll down to the IM area, which may be on this form directly or hidden under a "More" category:

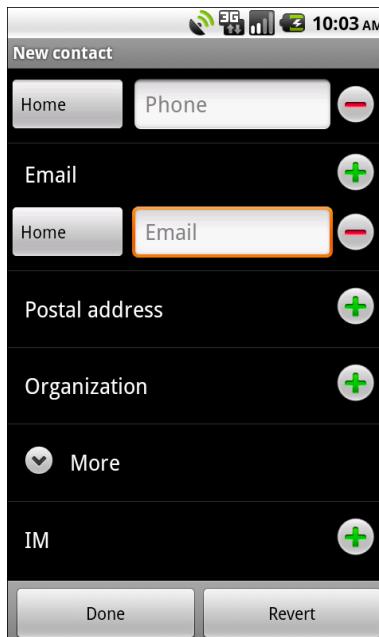


Figure 46. The contact detail form, scrolled to the "More" category, showing the IM option

Click the green plus button, which will add a blank IM set of widgets to your contact detail form:

Contacting Our Friends

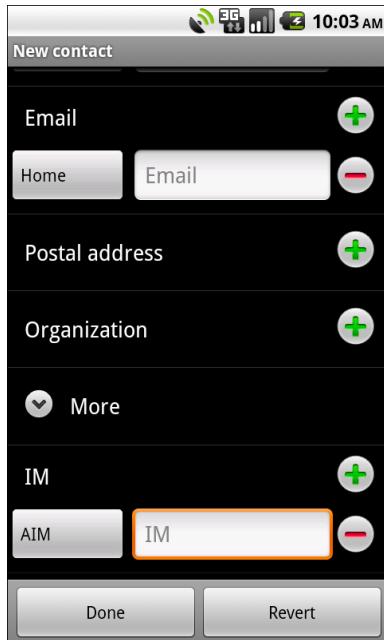


Figure 47. The contact detail form, showing the blank IM field

Click the AIM button to bring up the roster of available IM types:

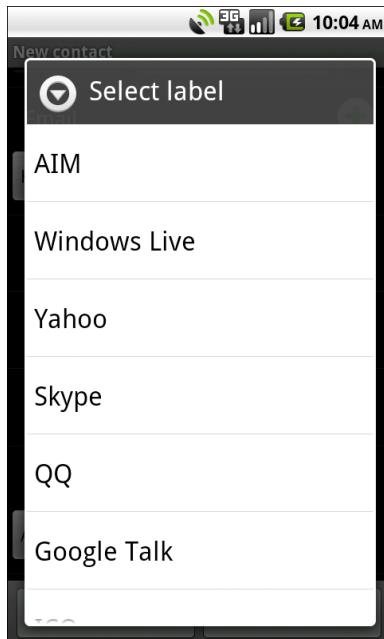


Figure 48. The dialog of IM types for contacts

Choose "Custom...", then fill in "identi.ca" (sans quotes) as the IM type:

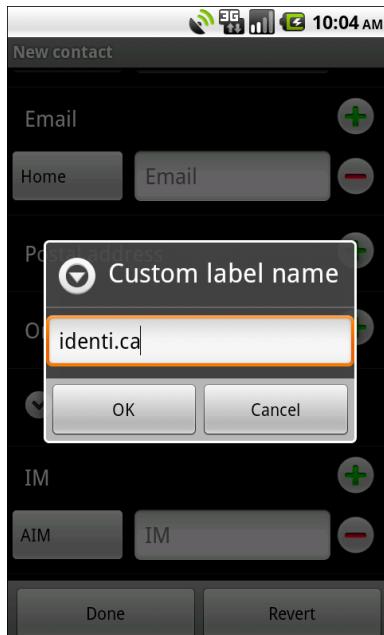


Figure 49. The IM type dialog, with "identi.ca" entered

Click OK, then fill in the identi.ca screen name for this contact in the IM field:

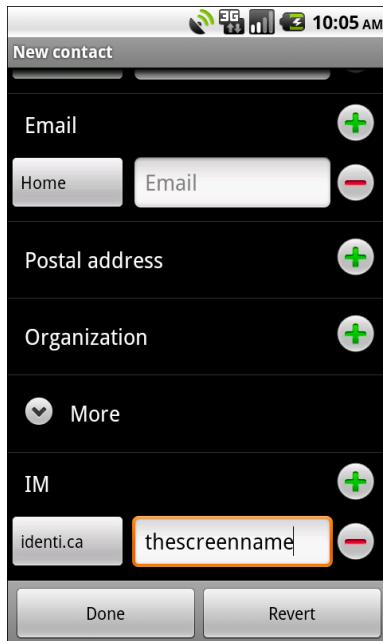


Figure 50. The contact detail form, showing the "identi.ca" IM type and "thescreenname" as the screen name

Click the Done button, and you are set. You will need one or more contacts set up with screen names. In particular, you need contacts that will have status updates in your timeline.

Step #2: Design the Highlight

Next, we need to decide how we wish to highlight those friends who are in our contacts database. A simple highlight that we can apply is to put a background color on the friend's name – that will help friends who are contacts to stand out from the rest.

To do that, we first need to amend `TimelineEntry`, so it knows whether or not it is a contact – add the following data member to the `TimelineEntry` inner class of `Patchy`:

```
boolean isContact=false;
```

Then, we need to modify `TimelineEntryWrapper` so it looks at the `isContact` property of `TimelineEntry` and sets the background accordingly, so change `populateFrom()` in `TimelineEntryWrapper` to be:

```
void populateFrom(TimelineEntry s) {
    getFriend().setText(s.friend);
    getCreatedAt().setText(s.createdAt);
    getStatus().setText(s.status);

    if (s.isContact) {
        getFriend().setBackgroundColor(0xFF0000FF);
    }
    else {
        getFriend().setBackgroundColor(0x00000000);
    }
}
```

You might think that we only need to set the background when the friend is a contact. However, since rows get recycled, if we do not reset the background when the friend is not a contact, soon all our rows will be highlighted, mostly in error.

Step #3: Find and Highlight Matching Contacts

Of course, our `isContact` property in `TimelineEntry` is always set to be `false`, which is not terribly helpful. Instead, we need to do a lookup to see if a friend is actually a contact, so we can set that flag appropriately. We do not need any data about the contact – all we need to know is if there exists a contact with the appropriate Twitter "organization" and screen name.

First, add the `READ_CONTACTS` permission to the `AndroidManifest.xml` file for `Patchy`. Without this, we cannot find out if a friend is a contact.

Next, add a `PROJECTION` to `Patchy`:

```
private static final String[] PROJECTION=new String[] {
    ContactsContract.Contacts._ID,
};
```

This just says that the only column we want back is the contact's ID. Also, add imports to `android.provider.ContactsContract` and `android.provider.ContactsContract.CommonDataKinds`.

Finally, modify the `TimelineEntry` constructor to be as follows:

```
TimelineEntry(String friend, String createdAt,
              String status) {
    this.friend=friend;
    this.createdAt=createdAt;
    this.status=status;

    String[] args={friend};
    StringBuilder query=new StringBuilder(CommonDataKinds.Im.CUSTOM_PROTOCOL);

    query.append("='identi.ca' AND ");
    query.append(CommonDataKinds.Im.DATA);
    query.append("=?");
    Cursor c=managedQuery(ContactsContract.Data.CONTENT_URI,
                          PROJECTION,
                          query.toString(),
                          args, null);

    if (c.getCount()>0) {
        this.isContact=true;
    }
}
```

We use `managedQuery()` to get a `Cursor` representing our contact, if any. All we do is look to see if we got one (or, conceivably, more) matches on our screen name, and use that information to set the value of `isContact`.

The net result is that any friends whose screen names are in our contacts will show up with their screen names on a blue background.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Provide a means from the `Patchy` UI to see the contact information for a contact from whom we have received a status update.
- Cache the matching contacts for a short period of time, to reduce contact lookups when new status updates come in.

- Provide a means from the Patchy UI to "re-tweet" both inside identi.ca and by forwarding the post (and its URL) to contacts via email or SMS.

Further Reading

Using content providers in general is covered in the "Using a Content Provider" chapter of [The Busy Coder's Guide to Android Development](#). Using the `ContactsContract` content provider specifically is covered in "The Contacts Content Provider", found in [The Busy Coder's Guide to Advanced Android Development](#).

Android Would Like Your Attention

From time to time, Android can tell you things that may be of interest, such as when the battery gets low. This allows you to take action based on those system events, such as reducing the amount of background work you do while the battery is low. In this tutorial, we will update the `PostMonitor` service to be gentler while the battery is low, by polling less frequently.

Step-By-Step Instructions

First, you need to have completed the previous `Patchy` tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the `28-UsingCP` edition of `Patchy` to use as a starting point.

Step #1: Track the Battery State

We need to register a `BroadcastListener` from `PostMonitor` to be informed of changes in state in the battery.

First, add the `registerReceiver()` call to `onCreate()` in `PostMonitor`:

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

```
new Thread(threadBody).start();
registerReceiver(onBatteryChanged,
    new IntentFilter(Intent.ACTION_BATTERY_CHANGED));
}
```

We set the receiver's IntentFilter to watch for ACTION_BATTERY_CHANGED events, routing them to an onBatteryChanged object.

You will need to add an import for android.content.IntentFilter.

Next, call unregisterReceiver() in onDestroy() in PostMonitor:

```
@Override
public void onDestroy() {
    super.onDestroy();

    unregisterReceiver(onBatteryChanged);
    active.set(false);
}
```

If we do not do this, our service will keep running and getting battery events even after it would ordinarily have shut down.

Both of these rely on an onBatteryChanged BroadcastReceiver implementation, which you should add to PostMonitor:

```
BroadcastReceiver onBatteryChanged=new BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
        int pct=100
            *intent.getIntExtra(BatteryManager.EXTRA_LEVEL, 1)
            /intent.getIntExtra(BatteryManager.EXTRA_SCALE, 1);

        isBatteryLow.set(pct<=25);
    }
};
```

All we do is examine the battery level (as measured on the battery level scale) and, if it hits 25% or lower, we set an isBatteryLow object to true; otherwise, we set it to false.

You will need to add imports for `android.os.BatteryManager`, `android.content.Context`, and `android.content.BroadcastReceiver`.

The `isBatteryLow` object is actually an `AtomicBoolean`, which you should add to the data members for `PostMonitor`:

```
private AtomicBoolean isBatteryLow=new AtomicBoolean(false);
```

At this point, we are watching for battery events and setting `isBatteryLow`, but we are not making use of that information anywhere.

Step #2: Use the Battery State

Then, all we need to do is look at `isBatteryLow` in our polling loop. Modify `threadBody` in `PostMonitor` to look like this:

```
private Runnable threadBody=new Runnable() {
    public void run() {
        while (active.get()) {
            for (Account l : accounts.values()) {
                poll(l);
            }

            int pollPeriod=POLL_PERIOD;

            if (isBatteryLow.get()) {
                pollPeriod*=10;
            }

            SystemClock.sleep(pollPeriod);
        }
    }
};
```

All we do is multiply our normal polling period by a factor of 10 when the battery is low.

To test this with an emulator, you will need to tell the emulator what charge level to simulate. To do this:

1. Determine the port number your emulator is using. This appears in the title bar of the emulator window, to the left of the colon. A typical value is 5554.
2. From a command prompt, run `telnet localhost 5554` (substituting in your port number for 5554). This will open a telnet prompt on your emulator.
3. Type `power capacity NN`, where `NN` is the battery charge level you want the emulator to emulate (e.g., 30 for 30%) and press <Enter>.
4. When you are done with the telnet session, type `quit` and press <Enter> to return to the command prompt.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Offer a user preference whereby `PostMonitor` will only poll if there is a WiFi connection available. Then, monitor the WiFi connection state and enable/disable polling as appropriate.
- Offer a user preference whereby `PostMonitor` will start collecting timeline updates on boot and will buffer some number of updates, so when `Patchy` connects, updates are available immediately and fewer are missed in between `Patchy` runs.

Further Reading

Additional coverage of Android-generated broadcast Intent objects can be found in the "Handling System Events" chapter of [The Busy Coder's Guide to Advanced Android Development](#).

Now, Your Friends Are Alarmed

One current flaw with `Patchy` is that if your device goes to sleep, you will not get status updates on your timeline. Clearly, this must be corrected.

In this tutorial you will have `PostMonitor` use `AlarmManager`, instead of its own polling loop, to wake it up as needed. You will also use a `WakeLock` to ensure that `PostMonitor` stays awake long enough to do a poll of the timeline. Fortunately, much of this is wrapped up in a reusable component that you will employ.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 29-SysEvents edition of `Patchy` to use as a starting point.

Step #1: Import a Reusable Component

Some guy wrote `WakefulIntentService`, which is covered in some detail in *The Busy Coder's Guide to Advanced Android Development*. `WakefulIntentService` is an ideal component to use in this application, to ensure that if the device wakes up in the middle of the night to poll Twitter for updates, that the device does not fall back asleep in the middle of doing that poll.

To obtain the JAR containing `WakefulIntentService`, you can either download the source code from its [GitHub repository](#) and run `ant jar` to build it, or you can download a pre-compiled JAR from that GitHub repository's "Downloads" area, or you can get the JAR out of the source code for this book in the `30-SysSvcs` edition of the `Patchy` project.

Either way, once you have the JAR, put it in your `libs/` directory.

Step #2: Create the Alarm BroadcastReceiver

In order to receive alarms from the `AlarmManager`, we need a `BroadcastReceiver` registered in our `AndroidManifest.xml` file. This `BroadcastReceiver`, in turn, needs to start our `PostMonitor` service, in case it is not running, and to let it know that a poll is required.

Create `Patchy/src/apt/tutorial/two/OnAlarmReceiver.java` with the following implementation:

```
package apt.tutorial.two;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;
import com.commonsware.cwac.wakeful.WakefulIntentService;

public class OnAlarmReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Intent i=new Intent(context, PostMonitor.class);
        i.setAction(PostMonitor.POLL_ACTION);
        WakefulIntentService.sendWakefulWork(context, i);
    }
}
```

Then, add it to `AndroidManifest.xml`, along with the `WAKE_LOCK` permission we need in order to use a `WakeLock`:

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

```
package="apt.tutorial.two"
    android:versionCode="1"
    android:versionName="1.0">
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.READ_CONTACTS" />
<uses-permission android:name="android.permission.WAKE_LOCK" />
<application android:label="@string/app_name">
    <uses-library android:name="com.google.android.maps" />
    <activity android:name=".Patchy"
        android:label="@string/app_name">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <activity android:name=".EditPreferences">
    </activity>
    <activity android:name=".StatusMap">
    </activity>
    <activity android:name=".HelpCast">
    </activity>
    <activity android:name=".HelpPage">
    </activity>
    <service android:name=".PostMonitor" />
    <receiver android:name=".OnAlarmReceiver">
    </receiver>
</application>
</manifest>
```

OnAlarmReceiver tells WakefulIntentService to sendWakefulWork(), which will:

- Acquire a static WakeLock, then
- Call startService() on our PostMonitor service

This means all WakeLock management can be encapsulated in WakefulIntentService.

Step #3: Doing the Work

Now, we need to arrange for PostMonitor to respond to those polling Intents.

First, change PostMonitor to have it extend WakefulIntentService, adding an import for com.commonsware.cwac.wakeful.WakefulIntentService.

Next, add a couple of private data members, one for an `AlarmManager` and one for a `PendingIntent`, with suitable imports:

```
private PendingIntent pi=null;
```

You will need to add an import for `android.app.AlarmManager`. The `AlarmManager` is the gateway to scheduled tasks in Android, while the `PendingIntent` will be invoked each time the alarm goes off.

Then, update `onCreate()` to obtain the `AlarmManager` system service and to set up the `PendingIntent` that `AlarmManager` should invoke every polling cycle, plus schedule the initial alarm:

```
public void onCreate() {
    super.onCreate();

    registerReceiver(onBatteryChanged,
                    new IntentFilter(Intent.ACTION_BATTERY_CHANGED));

    alarm=(AlarmManager) getSystemService(Context.ALARM_SERVICE);

    Intent i=new Intent(this, OnAlarmReceiver.class);

    pi=PendingIntent.getBroadcast(this, 0, i, 0);
    setAlarm(INITIAL_POLL_PERIOD);
}

@Override
```

Note that we also get rid of our statement to schedule the `threadBody` `Runnable` as a background thread, because an `IntentService` (parent class of `WakefulIntentService`) automatically lets us do our work on a background thread.

This requires a `setAlarm()` method to actually schedule the alarm:

```
private void setAlarm(long period) {
    alarm.set(AlarmManager.ELAPSED_REALTIME_WAKEUP,
              SystemClock.elapsedRealtime()+period,
              pi);
}
```

The `setAlarm()` implementation tells `AlarmManager` to schedule a one-shot alarm (rather than a recurring alarm), using the time-base of `SystemClock.elapsedRealtime()`, to go off in `period` milliseconds from now, invoking our `PendingIntent` at that point.

We now need to do the actual work itself. According to the protocol for `WakefulIntentService`, we need to override a `doWakefulWork()` method, which will be invoked inside a partial `WakeLock` every time the service is started. For those that are actual polling `Intents`, we want to poll the Twitter accounts, then schedule the next alarm:

```
@Override  
protected void doWakefulWork(Intent i) {  
    if (i.getAction().equals(POLL_ACTION)) {  
        for (Account l : accounts.values()) {  
            poll(l);  
        }  
        setAlarm(isBatteryLow.get() ? POLL_PERIOD*10 : POLL_PERIOD);  
    }  
}
```

You will recognize some of this functionality as mimicking the `threadBody` `Runnable` that we are no longer using.

Note that while `AlarmManager` supports recurring alarms, since our polling period may change based upon battery state, we are manually scheduling successive alarms here.

We also need to cancel the outstanding alarm when the service is destroyed:

```
@Override  
public void onDestroy() {  
    super.onDestroy();  
  
    alarm.cancel(pi);  
    unregisterReceiver(onBatteryChanged);  
}
```

Finally, `WakefulIntentService` requires a constructor, because `IntentService` (supplied by Android) does. The `WakefulIntentService`'s constructor takes a

String parameter that is the "name" of the service. What role this name plays is unclear. So, add a functional constructor:

```
public PostMonitor() {  
    super("PostMonitor");  
}
```

Now you can get rid of the `isActive` and `threadBody` data members, along with all references to them (which you will find in `onCreate()` and `onDestroy()`).

Now, if you rebuild and reinstall the project, Patchy should work as before. The exception is that if you have Patchy on a device, and you let the device fall asleep while Patchy is still active, you should still receive status updates.

Here is the entire listing of `PostMonitor` after all of these changes:

```
package apt.tutorial.two;  
  
import android.app.AlarmManager;  
import android.app.Notification;  
import android.app.NotificationManager;  
import android.app.PendingIntent;  
import android.app.Service;  
import android.content.BroadcastReceiver;  
import android.content.Context;  
import android.content.Intent;  
import android.content.IntentFilter;  
import android.os.BatteryManager;  
import android.os.Binder;  
import android.os.IBinder;  
import android.os.SystemClock;  
import android.util.Log;  
import java.util.HashSet;  
import java.util.List;  
import java.util.Map;  
import java.util.Set;  
import java.util.concurrent.ConcurrentHashMap;  
import java.util.concurrent.atomic.AtomicBoolean;  
import winterwell.jtwitter.Twitter;  
import apt.tutorial.IPostListener;  
import apt.tutorial.IPostMonitor;  
import com.commonsware.cwac.wakeful.WakefulIntentService;  
  
public class PostMonitor extends WakefulIntentService {  
    public static final int NOTIFICATION_ID=1337;  
    public static final String STATUS_UPDATE="apt.tutorial.three.STATUS_UPDATE";
```

```
public static final String FRIEND="apt.tutorial.three.FRIEND";
public static final String STATUS="apt.tutorial.three.STATUS";
public static final String CREATED_AT="apt.tutorial.three.CREATED_AT";
public static final String POLL_ACTION="apt.tutorial.three.POLL_ACTION";
private static final String NOTIFY_KEYWORD="snicklefritz";
private static final int INITIAL_POLL_PERIOD=1000;
private static final int POLL_PERIOD=60000;
private Set<Long> seenStatus=new HashSet<Long>();
private Map<IPostListener, Account> accounts=
    new ConcurrentHashMap<IPostListener, Account>();
private final Binder binder=new LocalBinder();
private AtomicBoolean isBatteryLow=new AtomicBoolean(false);
private AlarmManager alarm=null;
private PendingIntent pi=null;

public PostMonitor() {
    super("PostMonitor");
}

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

    registerReceiver(onBatteryChanged,
        new IntentFilter(Intent.ACTION_BATTERY_CHANGED));

    alarm=(AlarmManager) getSystemService(Context.ALARM_SERVICE);

    Intent i=new Intent(this, OnAlarmReceiver.class);

    pi=PendingIntent.getBroadcast(this, 0, i, 0);
    setAlarm(INITIAL_POLL_PERIOD);
}

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

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

    alarm.cancel(pi);
    unregisterReceiver(onBatteryChanged);
}

@Override
protected void doWakefulWork(Intent i) {
    if (i.getAction().equals(POLL_ACTION)) {
        for (Account l : accounts.values()) {
            poll(l);
        }
    }
}
```

```
        setAlarm(isBatteryLow.get() ? POLL_PERIOD*10 : POLL_PERIOD);
    }

private void setAlarm(long period) {
    alarm.set(AlarmManager.ELAPSED_REALTIME_WAKEUP,
              SystemClock.elapsedRealtime() + period,
              pi);
}

private void poll(Account l) {
    try {
        Twitter client = new Twitter(l.user, l.password);

        client.setAPIRootUrl("https://identi.ca/api");

        List<Twitter.Status> timeline = client.getFriendsTimeline();

        for (Twitter.Status s : timeline) {
            if (!seenStatus.contains(s.id)) {
                try {
                    Intent broadcast = new Intent(STATUS_UPDATE);
                    broadcast.putExtra(FRIEND, s.user.screenName);
                    broadcast.putExtra(STATUS, s.text);
                    broadcast.putExtra(CREATED_AT,
                                      s.createdAt.toString());
                    sendBroadcast(broadcast);
                }
                catch (Throwable t) {
                    Log.e("PostMonitor", "Exception in callback", t);
                }
                seenStatus.add(s.id);

                if (s.text.indexOf(NOTIFY_KEYWORD) > -1) {
                    showNotification();
                }
            }
        }
    }
    catch (Throwable t) {
        android.util.Log.e("PostMonitor",
                           "Exception in poll()", t);
    }
}

private void showNotification() {
    final NotificationManager mgr =
        (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
    Notification note = new Notification(R.drawable.status,
                                         "New matching post!",
                                         System.currentTimeMillis());
    Intent i = new Intent(this, Patchy.class);
}
```

```
i.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP |
            Intent.FLAG_ACTIVITY_SINGLE_TOP);

PendingIntent pi=PendingIntent.getActivity(this, 0,
                                             i,
                                             0);

note.setLatestEventInfo(this, "Identica Post!",
                       "Found your keyword: "+NOTIFY_KEYWORD,
                       pi);

mgr.notify(NOTIFICATION_ID, note);
}

BroadcastReceiver onBatteryChanged=new BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
        int pct=100
                *intent.getIntExtra(BatteryManager.EXTRA_LEVEL, 1)
                /intent.getIntExtra(BatteryManager.EXTRA_SCALE, 1);

        isBatteryLow.set(pct<=25);
    }
};

class Account {
    String user=null;
    String password=null;
    IPostListener callback=null;

    Account(String user, String password,
            IPostListener callback) {
        this.user=user;
        this.password=password;
        this.callback=callback;
    }
}

public class LocalBinder extends Binder implements IPostMonitor {
    public void registerAccount(String user, String password,
                               IPostListener callback) {
        Account l=new Account(user, password, callback);

        poll(l);
        accounts.put(callback, l);
    }

    public void removeAccount(IPostListener callback) {
        accounts.remove(callback);
    }
}
```

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Offer a user preference whereby `PostMonitor` will start collecting timeline updates on boot and will buffer some number of updates, so when `Patchy` connects, updates are available immediately and fewer are missed in between `Patchy` runs. This will require the initial alarm to be set from the on-boot receiver, rather than from the service's `onCreate()`.
- Extend the above extra credit component by offering a user preference to have `PostMonitor` raise a `Notification` if certain sorts of status updates are received while `Patchy` itself is *not* running.

Further Reading

You can learn more about the `AlarmManager` and the `WakefulIntentService` in the "Using System Services" chapter of [The Busy Coder's Guide to Advanced Android Development](#).

Searching For Food

In this tutorial, we will enable searching of the restaurant list, so you can find one in the vast array of restaurants you surely will maintain in this application.

Step-By-Step Instructions

First, you need to have completed the previous `LunchList` tutorial. If you are starting the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 14-Rotation edition of `LunchList` to use as a starting point.

Step #1: Have the List Conduct the Search

First, we need to be able to trigger the local search. To do this, add another item to our `LunchList/res/menu/options.xml` menu definition:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/add"
          android:title="Add"
          android:icon="@drawable/ic_menu_add"
        />
    <item android:id="@+id/search"
          android:title="Search"
          android:icon="@drawable/ic_menu_search"
        />
    <item android:id="@+id/prefs"
```

```
    android:title="Settings"
    android:icon="@drawable/ic_menu_preferences"
  />
</menu>
```

You will need a suitable icon, such as `ic_menu_search.png` from the Android SDK.

Then, add the corresponding segment to `onOptionsItemSelected()` in `LunchList` to handle our new item:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.add) {
        startActivity(new Intent(LunchList.this, DetailForm.class));

        return(true);
    }
    else if (item.getItemId()==R.id.prefs) {
        startActivity(new Intent(this, EditPreferences.class));

        return(true);
    }
    else if (item.getItemId()==R.id.search) {
        onSearchRequested();

        return(true);
    }

    return(super.onOptionsItemSelected(item));
}
```

This triggers `onSearchRequested()`, which tells Android that we would like to conduct a search in this application.

Finally, in `initList()` in `LunchList`, we need to see if our activity was launched via a search and, if so, use the search string entered by the user. Here is a revised `initList()` implementation for `LunchList` that does just that:

```
private void initList() {
    if (model!=null) {
        stopManagingCursor(model);
        model.close();
    }
}
```

```
String where=null;

if (Intent.ACTION_SEARCH.equals(getIntent().getAction())) {
    where="name LIKE \'%"+getIntent().getStringExtra(SearchManager.QUERY)+"%\'';
}

model=helper.getAll(where, prefs.getString("sort_order", "name"));
startManagingCursor(model);
adapter=new RestaurantAdapter(model);
setListAdapter(adapter);
}
```

The ACTION_SEARCH Intent will be sent to our activity if the activity was started as the result of a search request. Here, we are having the same activity be the "normal" presentation and display search results – this approach may or may not be appropriate for any given application. The search is simply a WHERE clause, searching for the search term (getIntent().getStringExtra(SearchManager.QUERY)) anywhere in the restaurant's name.

You will need to add an import for android.app.SearchManager.

This also means we need to accept a possible WHERE clause in the getAll() method of RestaurantHelper:

```
public Cursor getAll(String where, String orderBy) {
    StringBuilder buf=new StringBuilder("SELECT _id, name, address, type, notes
FROM restaurants");

    if (where!=null) {
        buf.append(" WHERE ");
        buf.append(where);
    }

    if (orderBy!=null) {
        buf.append(" ORDER BY ");
        buf.append(orderBy);
    }

    return(getReadableDatabase().rawQuery(buf.toString(), null));
}
```

Step #2: Integrate the Search in the Application

Now we need to tell Android that this application is searchable and how to do the search.

First, we need to overhaul our `AndroidManifest.xml` file to indicate that the `LunchList` activity is both searchable and the activity to launch to conduct a search:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="apt.tutorial"
    android:versionCode="1"
    android:versionName="1.0">
    <application android:label="@string/app_name">
        <activity android:name=".LunchList"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
            <intent-filter>
                <action android:name="android.intent.action.SEARCH" />
                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
            <meta-data android:name="android.app.searchable"
                android:resource="@xml/searchable" />
            <meta-data android:name="android.app.default_searchable"
                android:value=".LunchList" />
        </activity>
        <activity android:name=".DetailForm">
        </activity>
        <activity android:name=".EditPreferences">
        </activity>
    </application>
</manifest>
```

The second `<intent-filter>` indicates that the `LunchList` activity will handle any local search requests. The `android.app.searchable` metadata element names `@xml/searchable` as being the configuration details for the search itself. So, add a `LunchList/res/xml/searchable.xml` file with the following content:

```
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
    android:label="@string/searchLabel"
    android:hint="@string/searchHint" />
```

That file, in turn, references a pair of `String` resources, so we need to add them to `LunchList/res/values/strings.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">LunchList</string>
    <string name="searchLabel">Restaurants</string>
    <string name="searchHint">lorem</string>
</resources>
```

At this point, you can recompile and reinstall your application. Choosing the option menu will display our new search item:

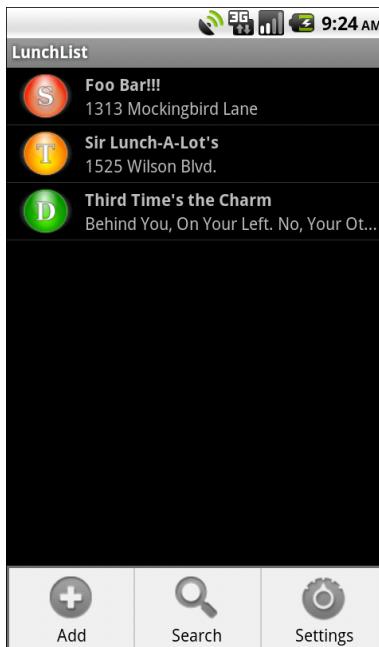


Figure 51. The new option menu

Choosing the search brings up the search field:

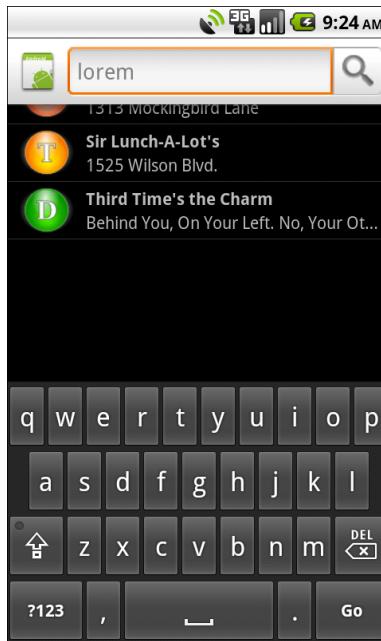


Figure 52. The search field

Entering in some value will give us a new `LunchList` with just the matching subset – those restaurants whose names contain the typed-in string.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Extend the search to search all relevant attributes of the data model, including address, notes, and the phone number.
- Support the Quick Search Box, by implementing another content provider, this one to provide search suggestions.

Further Reading

There is much more you can do to integrate with the Android search system. You will learn how in the "Searching with SearchManager" chapter of [The Busy Coder's Guide to Advanced Android Development](#).

Look Inside Yourself

In this tutorial, we will use the `PackageManager` to find out what operations are all possible on a contact pulled out of the contacts database on the device or emulator.

Step-By-Step Instructions

This tutorial starts a new application, independent from the `LunchList` and `Patchy` applications developed in the preceding tutorials.

Note that this tutorial will not work on the Android 2.2 emulator due to some sort of a bug, whereby contacts can be added but do not show up in the contacts list.

Step #1: Create a Stub Project

Using Eclipse or `android create project`, make a project named `Contacter` with a stub activity named `apt.tutorial.four.Contacter`. The generated activity class should resemble the following:

```
package apt.tutorial.four;  
  
import android.app.Activity;  
import android.os.Bundle;  
  
public class Contacter extends Activity  
{
```

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

Then, change it to be a `ListActivity` instead of an ordinary `Activity`, as we will be using some `ListActivity`-specific methods and support in this tutorial.

Step #2: Create a Layout

The goal of `Contacter` is to allow you to choose a contact, then show a list of possible actions on that contact via a `ListView`. That means we need a layout that lets us accomplish those ends.

With that in mind, create `Contacter/res/layout/main.xml` with the following content:

```
<?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"
    >
    <Button
        android:id="@+id/pick"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:text="Gimme a contact!"
        android:layout_weight="1"
    />
    <ListView
        android:id="@+id/list"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:layout_weight="1"
    />
</LinearLayout>
```

Step #3: Find the Correct Contact Uri

Since we want this particular code to run on a variety of Android releases, we need to figure out the right Uri to use to wrap in an ACTION_PICK Intent to let the user pick a contact. In Android 2.0 and newer, the Uri is android.provider.ContactsContract.Contacts.CONTENT_URI, while in Android 1.6 and earlier, the Uri is android.provider.Contacts.People.CONTENT_URI.

Fortunately, this value will not change during the execution of our application, so we can find out the right value via a static initializer, then use the value throughout the application.

Hence, add the following static data member and initializer block to Contacter, along with the required imports:

```
private static Uri CONTENT_URI=null;

static {
    int sdk=new Integer(Build.VERSION.SDK).intValue();

    if (sdk>=5) {
        try {
            Class clazz=Class.forName("android.provider.ContactsContract$Contacts");

            CONTENT_URI=(Uri)clazz.getField("CONTENT_URI").get(clazz);
        }
        catch (Throwable t) {
            Log.e("PickDemo", "Exception when determining CONTENT_URI", t);
        }
    }
    else {
        CONTENT_URI=Contacts.People.CONTENT_URI;
    }
}
```

Step #4: Attach the Button to the Contacts

Next, we want to arrange to let the user pick a contact when the button is clicked.

To do this, add the following implementation of `onCreate()` to Contacter:

```
public void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    if (CONTENT_URI==null) {
        Toast
            .makeText(this, "We are experiencing technical difficulties...",
                      Toast.LENGTH_LONG)
            .show();
        finish();

        return;
    }

    setContentView(R.layout.main);

    Button btn=(Button)findViewById(R.id.pick);

    btn.setOnClickListener(new View.OnClickListener() {
        public void onClick(View view) {
            Intent i=new Intent(Intent.ACTION_PICK, CONTENT_URI);

            startActivityForResult(i, PICK_REQUEST);
        }
    });
}
```

We first check to ensure our derived `CONTENT_URI` value is found; if not, we shut down the application, since there is nothing we can do. Otherwise, we load the layout, get the `Button`, and have it `startActivityForResult()` on the `ACTION_PICK` Intent on a button click.

We also need an implementation of `PICK_REQUEST`:

```
private static final int PICK_REQUEST=1337;
```

Step #5: Populate the List

We are not doing anything presently after the user makes their selection. What we want to do is display a roster of possible actions in the `ListView` in our layout.

First, create a layout to use with our rows, as `Contacter/res/layout/row.xml`:

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

```
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
  >
<ImageView android:id="@+id/icon"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentLeft="true"
    android:paddingLeft="2px"
    android:paddingTop="2px"
    android:paddingBottom="2px"
    android:paddingRight="5px"
  />
<TextView
    android:id="@+id/label"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_toRightOf="@+id/icon"
    android:textSize="11pt"
    android:paddingTop="2px"
    android:paddingBottom="2px"
  />
</RelativeLayout>
```

Then, let's implement `onActivityResult()` on `Contacter` to get the selected contact, get access to the `PackageManager`, craft an empty `Intent` on the selected contact, and query the `PackageManager` for available activities that work with that `Intent`. Finally, we sort the resulting list of `ResolveInfo` objects and pour them into an `ActionAdapter`:

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
                               Intent data) {
    if (requestCode==PICK_REQUEST) {
        if (resultCode==RESULT_OK) {
            Uri contact=data.getData();
            Intent stub=new Intent();

            stub.setData(contact);

            PackageManager pm=getPackageManager();
            List<ResolveInfo> actions=pm.queryIntentActivities(stub, 0);

            Collections.sort(actions,
                new ResolveInfo.DisplayNameComparator(pm));

            setListAdapter(new ActionAdapter(pm, actions));
        }
    }
}
```

ActionAdapter is a class we need to implement, to pour a ResolveInfo into the row layouts for our ListView. Hence, add the following ActionAdapter implementation as an inner class of Contacter:

```
class ActionAdapter extends ArrayAdapter<ResolveInfo> {
    private PackageManager pm=null;

    ActionAdapter(PackageManager pm, List<ResolveInfo> apps) {
        super(Contacter.this, R.layout.row, apps);
        this.pm=pm;
    }

    @Override
    public View getView(int position, View convertView,
                        ViewGroup parent) {
        if (convertView==null) {
            convertView=newView(parent);
        }

        bindView(position, convertView);

        return(convertView);
    }

    private View newView(ViewGroup parent) {
        return(LayoutInflater).inflate(R.layout.row, parent, false));
    }

    private void bindView(int position, View row) {
        TextView label=(TextView)row.findViewById(R.id.label);

        label.setText(getItem(position).loadLabel(pm));

        ImageView icon=(ImageView)row.findViewById(R.id.icon);

        icon.setImageDrawable(getItem(position).loadIcon(pm));
    }
}
```

At this point, you can compile and install your Contacter application. Initially, the list is empty, so the screen just shows the button:

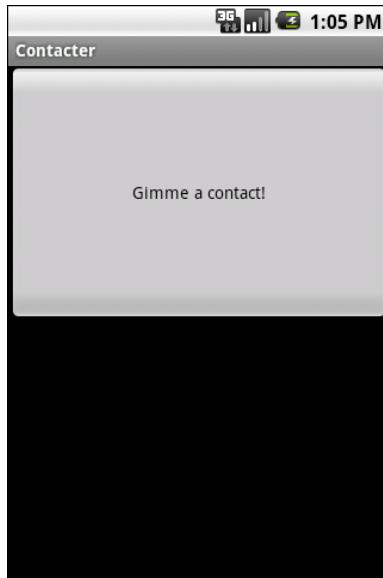


Figure 53. The Contacter application, as initially launched

Then, if you click the button and choose a contact, you will get a list of possible actions at the bottom:

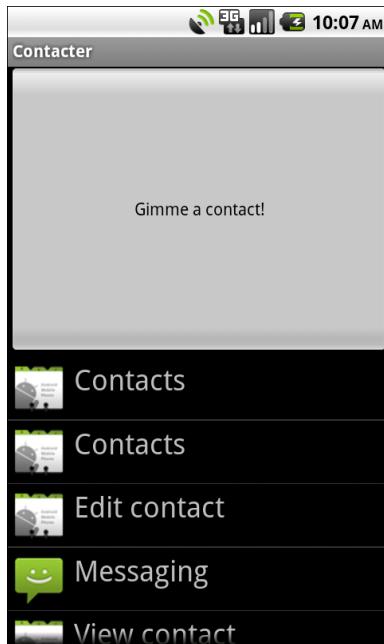


Figure 54. The Contacter application, after the user chooses a contact

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Arrange to get control on a list item click and display more details about the possible action, by getting details out of the `ResolveInfo` object (in particular, the `ActivityInfo` object held onto by the `ResolveInfo` object).
- Find a way to launch an actual activity from the information held in the `ResolveInfo` that starts up the desired action. You might want to take a peek at the source code to `addIntentOptions()` in the Android source code for ideas on how to accomplish this.

Further Reading

Other facets of `PackageManager` for introspection are covered in the "Introspection and Integration" chapter of [The Busy Coder's Guide to Advanced Android Development](#).

A Restaurant In Your Own Home

In this tutorial, we will create an "app widget", the term Android uses for interactive elements a user can add to their home screen. In particular, we will create an app widget that shows a random restaurant out of the `LunchList` database.

Step-By-Step Instructions

First, you need to have completed the previous `LunchList` tutorial. If you are starting the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the `31-Search` edition of `LunchList` to use as a starting point.

Step #1: Find An App Widget Background and Icon

We are going to need a background to use for our app widget, so its contents do not seem to float in empty space in the home screen. Ideally, this background is resizeable, so we have a choice of using an XML-defined drawable resource, or a nine-patch PNG.

If you examine the `33-AppWidget` project in the book's source code repository, you will see that there is a `widget_frame.9.png` file in `LunchList/res/drawable`. That nine-patch image works nicely for your app

widget. It is actually a clone of the nine-patch used as the background for the `Toast` class, culled from the Android open source project. You are welcome to use this image or find (or create) another of your choosing.

You will also need an icon, which will go alongside the `LunchList` name in Android's list of available widgets. Find some likely icon (32px square or so) and add it as `LunchList/res/drawable/icon.png`.

Step #2: Design the App Widget Layout

Next, we need to define a layout for our app widgets. App widgets are created via layout files, no different than activities, `ListView` rows, and the like. Right now, all we want is to show the name of the app widget, inside of something to serve as the widget's background.

So, create a `LunchList/res/layout/widget.xml` file with the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="@drawable/widget_frame"
>
    <TextView android:id="@+id/name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerVertical="true"
        android:layout_alignParentLeft="true"
        android:textSize="10pt"
        android:textColor="#FFFFFF"
    />
</RelativeLayout>
```

Step #3: Add an (Empty) AppWidgetProvider

Next, we need to create an `AppWidgetProvider`. `AppWidgetProvider`, a subclass of `BroadcastReceiver`, provides the base implementation for an app widget and gives us lifecycle methods like `onUpdate()` we can override to add custom behavior.

For now, though, just create an empty `AppWidgetProvider` implementation, with the truly unique name of `AppWidget`:

```
package apt.tutorial;  
  
import android.appwidget.AppWidgetProvider;  
  
public class AppWidget extends AppWidgetProvider {  
}
```

Step #4: Add the Widget Metadata

As part of wiring our app widget into our application, we need to create a "widget metadata" XML document. This file provides additional configuration definitions for the app widget, for things that cannot readily go into the manifest.

So, create a `LunchList/res/xml/widget_provider.xml` file with the following content:

```
<?xml version="1.0" encoding="utf-8"?>  
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"  
    android:minWidth="300dip"  
    android:minHeight="79dip"  
    android:updatePeriodMillis="1800000"  
    android:initialLayout="@layout/widget"  
/>
```

Here, we provide a height and width suggestion, which Android will convert into a number of "cells" given the actual screen size and density. Our height and width will give us a 4×1 cell widget, which means it will take up the entire width of a portrait mode screen.

The metadata also indicates the starting layout to use (the one we created earlier in this tutorial) and an "update period", which tells Android how frequently to ask us to update the app widget's contents (set to 30 minutes, in milliseconds).

Step #5: Update the Manifest

Now, we can add our widget to the manifest file. Edit `AndroidManifest.xml` to look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="apt.tutorial"
    android:versionCode="1"
    android:versionName="1.0">
    <application android:label="@string/app_name">
        <activity android:name=".LunchList"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
            <intent-filter>
                <action android:name="android.intent.action.SEARCH" />
                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
            <meta-data android:name="android.app.searchable"
                android:resource="@xml/searchable" />
            <meta-data android:name="android.app.default_searchable"
                android:value=".LunchList" />
        </activity>
        <activity android:name=".DetailForm">
        </activity>
        <activity android:name=".EditPreferences">
        </activity>
        <receiver android:name=".AppWidget"
            android:label="@string/app_name"
            android:icon="@drawable/icon">
            <intent-filter>
                <action
                    android:name="android.appwidget.action.APPWIDGET_UPDATE" />
            </intent-filter>
            <meta-data
                android:name="android.appwidget.provider"
                android:resource="@xml/widget_provider" />
        </receiver>
    </application>
</manifest>
```

In particular, note the `<receiver>` element towards the bottom – that is where we are teaching Android where our code and metadata resides for this app widget. The intent filter for `APPWIDGET_UPDATE` means that we will get control when Android wants us to update the app widget's contents, such as when the app widget is first added to the home screen.

At this point, you can compile and install the updated version of the application. Then, long-tap somewhere on the background of your home screen, to bring up the list of options for things to add to it:

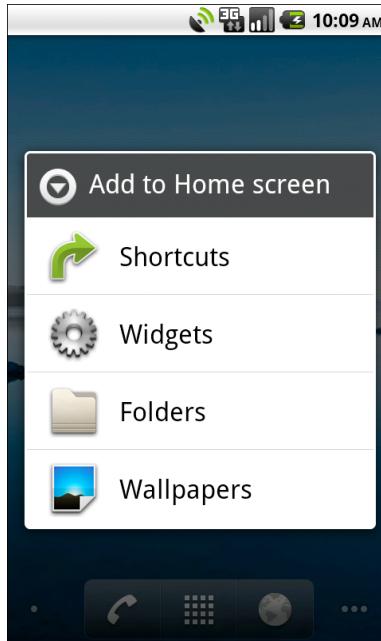


Figure 55. The list of things to add to the home screen

Choose Widgets, to bring up the list of available widgets:

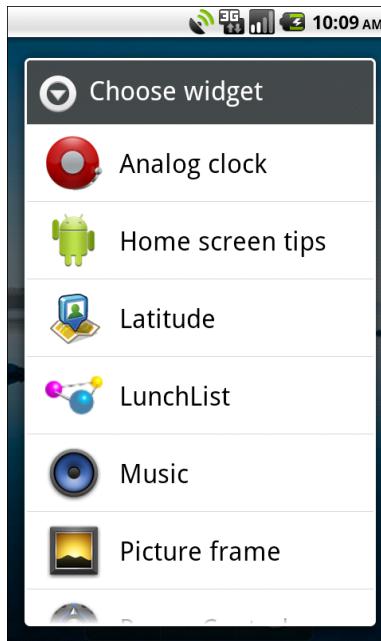


Figure 56. The list of available widgets

Then, choose our `LunchList` widget. It will show up, but have no contents, because we have not defined any contents yet:

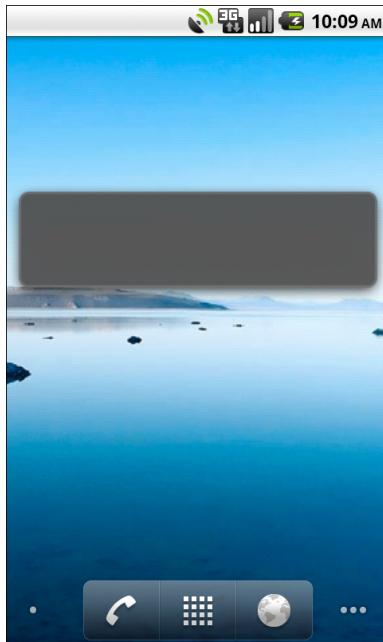


Figure 57. The very boring LunchList widget

Step #6: Show a Random Restaurant

Finally, we need to add some smarts that will actually display a random restaurant in the app widget. To do this, we will override the `onUpdate()` method in our `AppWidget` class and have it do the database I/O to find a random restaurant.

With that in mind, add the following `onUpdate()` implementation to `AppWidget`:

```
@Override  
public void onUpdate(Context ctxt,  
                     AppWidgetManager mgr,  
                     int[] appWidgetIds) {  
    ComponentName me=new ComponentName(ctxt, AppWidget.class);  
    RemoteViews updateViews=new RemoteViews("apt.tutorial",  
                                             R.layout.widget);  
    RestaurantHelper helper=new RestaurantHelper(ctxt);  
  
    try {  
        Cursor c=helper
```

```
        .getReadableDatabase()
        .rawQuery("SELECT COUNT(*) FROM restaurants", null);

    c.moveToFirst();

    int count=c.getInt(0);

    c.close();

    if (count>0) {
        int offset=(int)(count*Math.random());
        String args[]={String.valueOf(offset)};

        c=helper
            .getReadableDatabase()
            .rawQuery("SELECT name FROM restaurants LIMIT 1 OFFSET ?", args);
        c.moveToFirst();
        updateViews.setTextViewText(R.id.name, c.getString(0));
    }
    else {
        updateViews.setTextViewText(R.id.name,
                                    ctxt.getString(R.string.empty));
    }
}
finally {
    helper.close();
}

mgr.updateAppWidget(me, updateViews);
}
```

Here, we:

- Create a `RemoteViews` object, which represents a set of GUI "commands" to invoke on the home screen that defines how to modify the app widget
- Open up a database connection
- Find out how many restaurants there are via a SQL query
- Load a random restaurant via another SQL query
- Set the `name` `TextView` in the app widget (via the `RemoteViews`) to have either the name of the restaurant or an error message
- Update the app widget itself

You will also need to add a new string resource, named `empty`, that will go into the app widget if there is no restaurant available (e.g., the database is empty).

Make sure you have at least two restaurants in `LunchList` – we will need more than one to show the random effects, particularly starting with the next tutorial.

At this point, compile and reinstall the application. Also, if you got rid of the empty app widget, add a new one to your home screen. You should see the name of one of your restaurants:

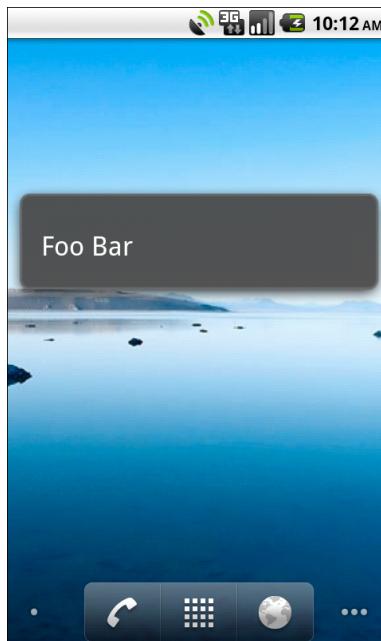


Figure 58. The app widget, showing the random restaurant

As app widgets go, this one is very unimpressive. However, we will make it somewhat better in the next tutorial.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Add other widgets to the app widget layout, such as a logo icon
- Reduce the font size of the name and add a second `TextView` to the layout to show the restaurant's address
- Experiment with other widget sizes instead of the `4x1` cell format used in the widget metadata

Further Reading

App widgets are covered in a chapter of [The Busy Coder's Guide to Advanced Android Development](#).

More Home Cooking

In this tutorial, we will add a few more features to our existing app widget, such as a button to choose another random restaurant. Also, if the user taps on the name of the restaurant, we will open up the detail form activity for that restaurant.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 33-AppWidget edition of `LunchList` to use as a starting point.

Step #1: Find a Button Graphic

Next, you will need an image to go on an `ImageButton` that will serve to change the restaurant shown by the app widget to another random restaurant.

If you examine the 34-AdvAppWidget project in the book's source code repository, you will see that there is a `ff.png` file there, culled from the Android open source project, that you could use. Or, substitute your own graphic as you see fit – just name it `ff.png` to match how it is used later in this tutorial.

Step #2: Add the Button to the Layout

Next, we want to modify the app widget layout to incorporate this `ImageButton`. Revise `LunchList/res/layout/widget.xml` to look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="@drawable/widget_frame"
>
    <TextView android:id="@+id/name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerVertical="true"
        android:layout_alignParentLeft="true"
        android:layout_toLeftOf="@+id/next"
        android:textSize="10pt"
        android:textColor="#FFFFFF"
    />
    <ImageButton android:id="@+id/next"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerVertical="true"
        android:layout_alignParentRight="true"
        android:src="@drawable/ff"
    />
</RelativeLayout>
```

At this point, if you compile and reinstall the application, then remove and re-add the app widget from your home screen, you will see the `ImageButton` appear:

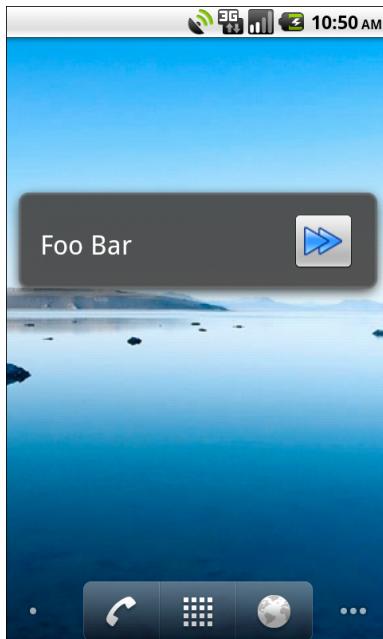


Figure 59. The app widget with the newly-added button

Step #3: Migrate Update Logic to an IntentService

Right now, we are doing our database I/O right in our `AppWidgetProvider` implementation's `onUpdate()` method. That is probably fine, but if we add too much logic, we run the risk of timing out, as `onUpdate()` is called on the main application thread. Since we can update our app widget asynchronously without issue, it is safer if we can delegate this work to something that can be done off the main thread, such as an `IntentService`.

With that in mind, create an `IntentService` named `WidgetService` in the `LunchList` project, with the following implementation:

```
package apt.tutorial;

import android.app.IntentService;
import android.app.PendingIntent;
import android.appwidget.AppWidgetManager;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
```

```
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.widget.RemoteViews;

public class WidgetService extends IntentService {
    public WidgetService() {
        super("WidgetService");
    }

    @Override
    public void onHandleIntent(Intent intent) {
        ComponentName me=new ComponentName(this, AppWidget.class);
        RemoteViews updateViews=new RemoteViews("apt.tutorial",
            R.layout.widget);
        RestaurantHelper helper=new RestaurantHelper(this);
        AppWidgetManager mgr=AppWidgetManager.getInstance(this);

        try {
            Cursor c=helper
                .getReadableDatabase()
                .rawQuery("SELECT COUNT(*) FROM restaurants", null);

            c.moveToFirst();

            int count=c.getInt(0);

            c.close();

            if (count>0) {
                int offset=(int)(count*Math.random());
                String args[]={String.valueOf(offset)};

                c=helper
                    .getReadableDatabase()
                    .rawQuery("SELECT _ID, name FROM restaurants LIMIT 1 OFFSET ?",
args);
                c.moveToFirst();
                updateViews.setTextViewText(R.id.name, c.getString(0));
            }
            else {
                updateViews.setTextViewText(R.id.title,
                    this.getString(R.string.empty));
            }
        }
        finally {
            helper.close();
        }

        mgr.updateAppWidget(me, updateViews);
    }
}
```

You will notice that the `onHandleIntent()` method is almost identical to the current `onUpdate()` method in `AppWidget`. The differences are that all references to `ctxt` are replaced with `this` (since the `IntentService` is a `Context`) and that we need to obtain an `AppWidgetManager` rather than use one passed into us. But, since `onHandleIntent()` is run on a background thread, we can take as much time as is necessary.

Then, add `WidgetService` to the manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="apt.tutorial"
    android:versionCode="1"
    android:versionName="1.0">
    <application android:label="@string/app_name">
        <activity android:name=".LunchList"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
            <intent-filter>
                <action android:name="android.intent.action.SEARCH" />
                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
            <meta-data android:name="android.app.searchable"
                android:resource="@xml/searchable" />
            <meta-data android:name="android.app.default_searchable"
                android:value=".LunchList" />
        </activity>
        <activity android:name=".DetailForm">
        </activity>
        <activity android:name=".EditPreferences">
        </activity>
        <receiver android:name=".AppWidget"
            android:label="@string/app_name"
            android:icon="@drawable/icon">
            <intent-filter>
                <action
                    android:name="android.appwidget.action.APPWIDGET_UPDATE" />
            </intent-filter>
            <meta-data
                android:name="android.appwidget.provider"
                android:resource="@xml/widget_provider" />
        </receiver>
        <service android:name=".WidgetService" />
    </application>
</manifest>
```

Finally, remove all of the code from the `onUpdate()` implementation in `AppWidget`, and replace it with a single call to `startService()`:

```
package apt.tutorial;

import android.appwidget.AppWidgetManager;
import android.appwidget.AppWidgetProvider;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.database.Cursor;
import android.widget.RemoteViews;
import android.database.sqlite.SQLiteDatabase;

public class AppWidget extends AppWidgetProvider {
    @Override
    public void onUpdate(Context ctxt,
                         AppWidgetManager mgr,
                         int[] appWidgetIds) {
        ctxt.startService(new Intent(ctxt, WidgetService.class));
    }
}
```

At this point, if you rebuild and reinstall the project, you will see that things work as they did before, despite this fairly radical implementation shift.

Step #4: Get Control on Button Clicks

Now, we need to arrange to execute some code when the user presses our new button. Specifically, we want to simply update the app widget itself, so we get a new random restaurant.

Since we pulled our updating logic into the `WidgetService`, we need to have that button call `startService()` to request `WidgetService` to do another update. However, this is an app widget, so we cannot simply add an `OnClickListener` to the button – the button is not in our code, but is in the home screen, configured via the `RemoteViews` object.

Instead, we can register a `PendingIntent`, to tell Android what to do when the button is clicked.

Add the following lines to the end of the `onHandleIntent()` method in `WidgetService`:

```
Intent i=new Intent(this, WidgetService.class);
PendingIntent pi=PendingIntent.getService(this, 0, i, 0);

updateViews.setOnClickListener(R.id.next, pi);
mgr.updateAppWidget(me, updateViews);
```

Here, we create a service `PendingIntent` and register it as the click handler for the button.

Now, if you recompile and reinstall the project, you will see that pressing the button gives you a new restaurant (unless it happens to randomly select the current one again).

Step #5: Get Control on Name Clicks

Finally, we want to bring up the `DetailForm` activity on the currently-visible restaurant if the user taps on the restaurant's name. To do this, we will need another `PendingIntent`, but we also need a bit more data from the content provider first.

Replace the current implementation of `onHandleIntent()` in `WidgetService` with the following:

```
@Override
public void onHandleIntent(Intent intent) {
    ComponentName me=new ComponentName(this, AppWidget.class);
    RemoteViews updateViews=new RemoteViews("apt.tutorial",
                                             R.layout.widget);
    RestaurantHelper helper=new RestaurantHelper(this);
    AppWidgetManager mgr=AppWidgetManager.getInstance(this);

    try {
        Cursor c=helper
            .getReadableDatabase()
            .rawQuery("SELECT COUNT(*) FROM restaurants", null);

        c.moveToFirst();

        int count=c.getInt(0);
```

```
c.close();

if (count>0) {
    int offset=(int)(count*Math.random());
    String args[]={String.valueOf(offset)};

    c=helper
        .getReadableDatabase()
        .rawQuery("SELECT _ID, name FROM restaurants LIMIT 1 OFFSET ?", args);
    c.moveToFirst();
    updateViews.setTextViewText(R.id.name, c.getString(1));

    Intent i=new Intent(this, DetailForm.class);

    i.putExtra(LunchList.ID_EXTRA, c.getString(0));

    PendingIntent pi=PendingIntent.getActivity(this, 0, i,
                                                PendingIntent.FLAG_UPDATE_CURRENT);

    updateViews.setOnClickListener(R.id.name, pi);
}
else {
    updateViews.setTextViewText(R.id.title,
                                this.getString(R.string.empty));
}
finally {
    helper.close();
}

Intent i=new Intent(this, WidgetService.class);
PendingIntent pi=PendingIntent.getService(this, 0, i, 0);

updateViews.setOnClickListener(R.id.next, pi);
mgr.updateAppWidget(me, updateViews);
}
```

You will notice a few changes:

- We now get the `_ID` column out of our content provider, in addition to the name, and we adjust the code that fills in the name widget's text to match.
- We create an activity `PendingIntent` for the `DetailForm` activity. However, we add in the `ID_EXTRA` "extra", so `DetailForm` knows which restaurant to show. And, since we are changing extras on an otherwise-unchanging `Intent`, we need to add the `FLAG_UPDATE_CURRENT` flag in our `getActivity()` call, so the new extra takes effect

- We then attach the `PendingIntent` to the name widget in the app widget

Now, if you recompile and reinstall the project, you will see that pressing the name of the restaurant brings up the restaurant's `DetailForm`.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Rather than show a random restaurant, keep track of the last restaurant viewed and cycle through them in progression, looping back to the first in the list when you reach the end. Consider adding a second `ImageButton` to move backwards through the list.
- Add another button that, when clicked, displays a `Toast` of the notes for the currently-viewed restaurant.

Further Reading

App widgets are covered in a chapter of [The Busy Coder's Guide to Advanced Android Development](#).

Take a Monkey to Lunch

In this tutorial, we will use the Monkey utility to stress test the `LunchList` application.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 34-AdvAppWidget edition of `LunchList` to use as a starting point.

Step #1: Prep `LunchList`

Ensure your `LunchList` has a few restaurants, of different types. Then, leave the `LunchList` at the `LunchList` activity itself (i.e., the list of available restaurants).

Step #2: Run the Monkey

Launch a command prompt or shell, and run the following command:

```
adb shell monkey -p apt.tutorial -v --throttle 100 600
```

Note that if you did not add your SDK's tools/ directory to your system PATH, you may need to change to that directory to get this command to execute properly.

This command indicates:

- You want to run the Monkey
- You want the Monkey to limit itself to testing your application (-p apt.tutorial), so if the Monkey attempts to do something that would exit your application (e.g., click the HOME button), that simulated input will be skipped
- You want the Monkey to execute one event every 100 milliseconds (--throttle 100)
- You want the Monkey to be verbose and report what events it simulates (-v)
- You want the Monkey to perform 600 simulated events

What you should see is the `LunchList` application running amok, as if some monkey were trying out different UI operations (clicking buttons, typing in fields, choosing menu options). If all goes well, `LunchList` will survive without errors. If something goes wrong, you will get an exception, and can use the log information (via DDMS or `adb logcat`) to see what failed and, possibly, how to fix it.

Your shell will show a running tally of what has been done, such as simulating screen taps or key presses.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Try updating your test process to be repeatable, so if you encounter some sort of exception, you can make it happen again. To do this, you will need to save your database file (stored in `/data/data/apt.tutorial/databases/lunchlist.db`) before running Monkey with the `-s` switch to provide a known seed value. Each test

run should back up the database, run Monkey with a fresh seed, and restore the database. If you got a crash or some other problem, re-run the process with the same seed, and you should be able to reproduce the failure.

- Experiment with additional options to configure the Monkey's operation, as described in the [Monkey](#) documentation.
- Experiment with Android's built-in copy of the JUnit test framework to exercise the restaurant model class programmatically.

Further Reading

More about Android's test-related features, including more on the Monkey, can be found in the "Testing" chapter of [*The Busy Coder's Guide to Advanced Android Development*](#).

Asking Permission to Place a Call

In this tutorial, we will add a bit of code that asks permission to place a call, and we will add a phone number to our restaurant data model and detail form. Then, we will actually place the call.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 35-AdvAppWidget edition of `LunchList` to use as a starting point.

Step #1: Add a Phone Number to the Database Schema

If we want our phone numbers to stick around, we need to put them in the database.

With that in mind, update `RestaurantHelper` to use the following implementation of `onCreate()`:

```
@Override  
public void onCreate(SQLiteDatabase db) {
```

```
    db.execSQL("CREATE TABLE restaurants (_id INTEGER PRIMARY KEY AUTOINCREMENT,  
        name TEXT, address TEXT, type TEXT, notes TEXT, phone TEXT);");  
}
```

Any time you make a material modification to the schema, you also need to increment the schema version number. For RestaurantHelper, that is held in SCHEMA_VERSION, so increment to 2:

```
private static final int SCHEMA_VERSION=2;
```

Step #2: Intelligently Handle Database Updates

When the schema version increments, onUpgrade() is called on RestaurantHelper rather than onCreate(). It is our job to update the schema, preferably without losing any user data. Here, we just use an ALTER TABLE SQL statement, since all we are doing is adding a column:

```
@Override  
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {  
    if (oldVersion==1 && newVersion==2) {  
        db.execSQL("ALTER TABLE restaurants ADD phone TEXT;");  
    }  
}
```

Step #3: Add Phone Number Support to the Rest of the Helper

We also need to update our insert() method on RestaurantHelper to accept a phone number:

```
public void insert(String name, String address,  
                  String type, String notes,  
                  String phone) {  
    ContentValues cv=new ContentValues();  
  
    cv.put("name", name);  
    cv.put("address", address);  
    cv.put("type", type);  
    cv.put("notes", notes);  
    cv.put("phone", phone);  
  
    getWritableDatabase().insert("restaurants", "name", cv);  
}
```

and the corresponding update() method, also to accept a phone number:

```
public void update(String id, String name, String address,
                   String type, String notes, String phone) {
    ContentValues cv=new ContentValues();
    String[] args={id};

    cv.put("name", name);
    cv.put("address", address);
    cv.put("type", type);
    cv.put("notes", notes);
    cv.put("phone", phone);

    getWritableDatabase().update("restaurants", cv, "_ID=?",
                               args);
}
```

The two query methods, getAll() and getById(), also should return the phone number from each of their respective queries:

```
public Cursor getAll(String where, String orderBy) {
    StringBuilder buf=new StringBuilder("SELECT _id, name, address, type, notes,"
                                         "phone FROM restaurants");

    if (where!=null) {
        buf.append(" WHERE ");
        buf.append(where);
    }

    if (orderBy!=null) {
        buf.append(" ORDER BY ");
        buf.append(orderBy);
    }

    return(getReadableDatabase().rawQuery(buf.toString(), null));
}

public Cursor getById(String id) {
    String[] args={id};

    return(getReadableDatabase()
           .rawQuery("SELECT _id, name, address, type, notes, phone FROM"
           "restaurants WHERE _ID=?",
           args));
}
```

Step #4: Collect the Phone Number on the Detail Form

If we actually want to have phone numbers, though, we need to actually collect them on `DetailForm`.

First, update `LunchList/res/layout/detail_form.xml` to add the following after the address row in our `TableLayout`:

```
<TableRow>
    <TextView android:text="Phone:" />
    <EditText android:id="@+id/phone" android:inputType="phone" />
</TableRow>
```

Notice that we are using `android:inputType="phone"` on the new `EditText` widget. This will cause Android to use a soft keyboard set up for entering a phone number (where available), rather than a standard keyboard layout.

Similarly, add the following after the address row in `LunchList/res/layout-land/detail_form.xml`:

```
<TableRow>
    <TextView android:text="Phone:" />
    <EditText android:id="@+id/phone"
              android:inputType="phone"
              android:layout_span="3"
    />
</TableRow>
```

Then, as in the previous section, clone all references to `address` in `DetailForm` to make references to our phone widgets, such as:

```
EditText address=null;
EditText phone=null;
```

and:

```
address=(EditText)findViewById(R.id.addr);
phone=(EditText)findViewById(R.id.phone);
```

Also, it is safe for you to go ahead and get rid of the implementations of `onSaveInstanceState()` and `onRestoreInstanceState()`.

At this point, you can recompile and reinstall the application. When you first run it, there will be a tiny pause as the database is updated. After that point, you can use the new field to add phone numbers to whichever restaurants you want:

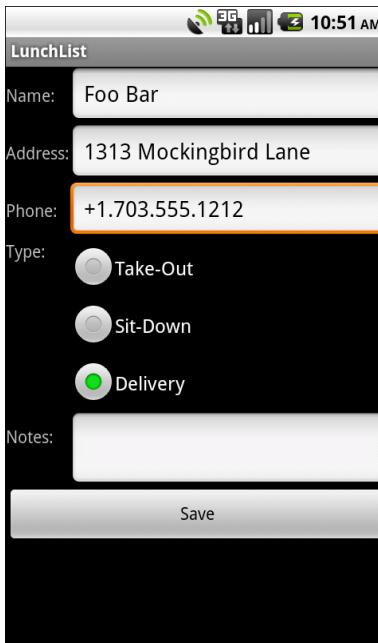


Figure 60. The new DetailForm layout

Step #5: Ask for Permission to Make Calls

Then, we can update `AndroidManifest.xml` to put in a permission request to be able to place phone calls:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="apt.tutorial"
    android:versionCode="1"
    android:versionName="1.0">
    <uses-permission android:name="android.permission.CALL_PHONE"/>
    <application android:label="@string/app_name">
        <activity android:name=".LunchList"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
```

```
<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
<intent-filter>
    <action android:name="android.intent.action.SEARCH" />
    <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
<meta-data android:name="android.app.searchable"
    android:resource="@xml/searchable" />
<meta-data android:name="android.app.default_searchable"
    android:value=".LunchList" />
</activity>
<activity android:name=".DetailForm">
</activity>
<activity android:name=".EditPreferences">
</activity>
<receiver android:name=".AppWidget"
    android:label="@string/app_name"
    android:icon="@drawable/icon">
    <intent-filter>
        <action
            android:name="android.appwidget.action.APPWIDGET_UPDATE" />
    </intent-filter>
    <meta-data
        android:name="android.appwidget.provider"
        android:resource="@xml/widget_provider" />
</receiver>
<service android:name=".WidgetService" />
</application>
</manifest>
```

Step #6: Dial the Number

Next, let us set up `DetailForm` with its own option menu that contains a `Call` item. When chosen, we dial the phone number, assuming there is one.

First, create `LunchList/res/menu/option_detail.xml` with the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/call"
        android:title="Call"
        android:icon="@drawable/ic_menu_call"
    />
</menu>
```

Then, add the following methods to `DetailForm`:

```
@Override  
public boolean onCreateOptionsMenu(Menu menu) {  
    new MenuInflater(getApplicationContext())  
        .inflate(R.menu.option_detail, menu);  
  
    return(super.onCreateOptionsMenu(menu));  
}  
  
@Override  
public boolean onOptionsItemSelected(MenuItem item) {  
    if (item.getItemId()==R.id.call) {  
        String toDial="tel:"+phone.getText().toString();  
  
        if (toDial.length()>4) {  
            startActivity(new Intent(Intent.ACTION_DIAL,  
                Uri.parse(toDial)));  
        }  
    }  
  
    return(super.onOptionsItemSelected(item));  
}
```

Note that you will need to add a number of imports (`Intent`, `Menu`, `MenuInflater`, `MenuItem`, and `Uri`) to get this to compile cleanly.

In the new code, we check to see if there is a phone number. If so, we wrap the phone number in a `tel:` `Uri`, then put that in an `ACTION_DIAL` `Intent` and start an activity on that `Intent`. This puts the phone number in the dialer.

If you rebuild and reinstall the application and try out the new menu choice on some restaurant with a phone number, you will see the `Dialer` appear:

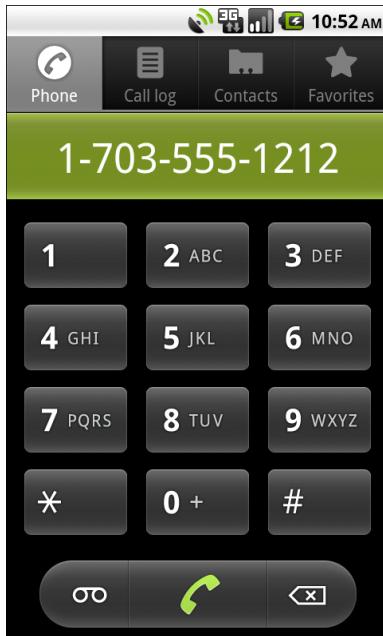


Figure 61. The Dialer

Step #7: Make the Call

Suppose we want to take advantage of the `CALL_PHONE` permission we requested earlier in this tutorial. All that we need to do is switch our Intent from `ACTION_DIAL` to `ACTION_CALL`:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.call) {
        String toDial="tel:"+phone.getText().toString();

        if (toDial.length(>4) {
            startActivity(new Intent(Intent.ACTION_CALL,
                Uri.parse(toDial)));

            return(true);
        }
    }

    return(super.onOptionsItemSelected(item));
}
```

Now, if you rebuild and reinstall the application, and try choosing the Call option menu item, you will immediately "call" the phone number...which will actually place a phone call if you are trying this on a device. The emulator, of course, cannot place phone calls.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Add a preference to display the phone number instead of the address in the restaurant list. Have the list detect the preference and fill in the second line of the restaurant rows accordingly.
- Push your APK file to a Web site that is configured to support the proper MIME type for Android application downloads (e.g., Amazon S3). Try installing your APK onto a device from the published location, to see how your requested permission appears to end users at install time.
- Find a revision to the `layout-land` version of `detail_form.xml` that does not clip the bottom radio button.

Further Reading

Permissions in general are covered in the "Requesting and Requiring Permissions" chapter of [The Busy Coder's Guide to Android Development](#). Working with the telephony features of Android is briefly covered in the "Handling Telephone Calls" chapter of the same book.

Photographic Memory

One logical thing to add to the restaurant information in `LunchList` would be photos: the exterior of the restaurant, favorite dishes, wait staff to avoid, etc. This tutorial will bring us partway there, by allowing users to take a picture while in `LunchList`.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 36-PermsPhone edition of `LunchList` to use as a starting point.

Step #1: Adjust the Manifest

To date, we have been able to skate by without a number of elements in our manifest that many applications need. At this point, though, we need to make some adjustments. So, make your `LunchList/AndroidManifest.xml` file look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="apt.tutorial"
    android:versionCode="1"
    android:versionName="1.0">
    <uses-sdk
        android:minSdkVersion="3"
```

```
        android:targetSdkVersion="7"
    />
<supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false"
/>
<uses-feature android:name="android.hardware.camera" />
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.CALL_PHONE"/>
<application android:label="@string/app_name">
    <activity android:name=".LunchList"
        android:label="@string/app_name">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
        <intent-filter>
            <action android:name="android.intent.action.SEARCH" />
            <category android:name="android.intent.category.DEFAULT" />
        </intent-filter>
        <meta-data android:name="android.app.searchable"
            android:resource="@xml/searchable" />
        <meta-data android:name="android.app.default_searchable"
            android:value=".LunchList" />
    </activity>
    <activity android:name=".DetailForm">
    </activity>
    <activity android:name=".EditPreferences">
    </activity>
    <activity android:name=".Photographer"
        android:configChanges="keyboardHidden|orientation"
        android:screenOrientation="landscape"
        android:theme="@android:style/Theme.NoTitleBar.Fullscreen">
    </activity>
    <receiver android:name=".AppWidget"
        android:label="@string/app_name"
        android:icon="@drawable/icon">
        <intent-filter>
            <action
                android:name="android.appwidget.action.APPWIDGET_UPDATE" />
        </intent-filter>
        <meta-data
            android:name="android.appwidget.provider"
            android:resource="@xml/widget_provider" />
    </receiver>
    <service android:name=".WidgetService" />
</application>
</manifest>
```

You will need to add:

- The `<uses-sdk>` element, indicating that while the code should run on Android 1.5 and newer (`android:minSdkVersion="3"`), that we are targeting Android 2.1 (`android:targetSdkVersion="7"`).
- The `<supports-screens>` element, indicating that our application supports both large (4" or higher) and normal (3-4") screens, but not small screens.
- The `<uses-feature>` element, indicating that we are going to be using the camera from this point forward. This will tell the system to prevent installation of our application on devices that lack a camera.
- Another `<uses-permission>` element, this time to tell the user that we want to use the camera.
- An `<activity>` element for a `Photographer` class, that we will be adding shortly. Note that we set `Photographer` to be always landscape and use a theme that makes it full-screen, removing the title bar and the status bar.

Step #2: Create the Photographer Layout

We need a layout with a full-screen `SurfaceView` to use as the space to display the "preview" – what the camera currently sees.

Create `LunchList/res/layout/photographer.xml` with the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<android.view.SurfaceView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/preview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
/>
```

This is merely a full-screen `SurfaceView`, the space on which the camera will draw its preview frames.

Step #3: Create the Photographer Class

Next, we need a class that will both connect the live preview to the SurfaceView, plus intercept the camera button for use in taking an actual picture. To keep things simple, we will hold off for now on doing anything "for real" with the actual picture.

With that in mind, add a `Photographer` class that looks like the following:

```
package apt.tutorial;

import android.app.Activity;
import android.graphics.PixelFormat;
import android.hardware.Camera;
import android.os.Bundle;
import android.os.Environment;
import android.util.Log;
import android.view.KeyEvent;
import android.view.SurfaceHolder;
import android.view.SurfaceView;
import android.widget.Toast;

public class Photographer extends Activity {
    private SurfaceView preview=null;
    private SurfaceHolder previewHolder=null;
    private Camera camera=null;

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

        preview=(SurfaceView)findViewById(R.id.preview);
        previewHolder=preview.getHolder();
        previewHolder.addCallback(surfaceCallback);
        previewHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
    }

    @Override
    public boolean onKeyDown(int keyCode, KeyEvent event) {
        if (keyCode==KeyEvent.KEYCODE_CAMERA ||
            keyCode==KeyEvent.KEYCODE_SEARCH) {
            takePicture();

            return(true);
        }

        return(super.onKeyDown(keyCode, event));
    }
}
```

```
private void takePicture() {
    camera.takePicture(null, null, photoCallback);
}

SurfaceHolder.Callback surfaceCallback=new SurfaceHolder.Callback() {
    public void surfaceCreated(SurfaceHolder holder) {
        camera=Camera.open();

        try {
            camera.setPreviewDisplay(previewHolder);
        }
        catch (Throwable t) {
            Log.e("Photographer",
                  "Exception in setPreviewDisplay()", t);
            Toast
                .makeText(Photographer.this, t.getMessage(),
                          Toast.LENGTH_LONG)
                .show();
        }
    }

    public void surfaceChanged(SurfaceHolder holder,
                               int format, int width,
                               int height) {
        Camera.Parameters parameters=camera.getParameters();

        parameters.setPreviewSize(width, height);
        parameters.setPictureFormat(PixelFormat.JPEG);

        camera.setParameters(parameters);
        camera.startPreview();
    }

    public void surfaceDestroyed(SurfaceHolder holder) {
        camera.stopPreview();
        camera.release();
        camera=null;
    }
};

Camera.PictureCallback photoCallback=new Camera.PictureCallback() {
    public void onPictureTaken(byte[] data, Camera camera) {
        // do something with the photo JPEG (data[]) here!
        camera.startPreview();
    }
};
```

Here, we initialize our `SurfaceView` in `onCreate()`. When the `SurfaceView` has been created, we connect the camera to it. When the `SurfaceView` size has been set (in `surfaceChanged()`), we configure the `Camera` to work with the

size of the `SurfaceView`. When the camera button is clicked, we take a picture, re-enabling the preview in the `PhotoCallback`.

Step #4: Tie In the Photographer Class

Finally, we need to allow the user to take a picture. Since we (theoretically) want our photos taken with `Photographer` to be associated with the restaurant, we can add an option menu choice to our `DetailForm`. Here is the revised `LunchList/res/menu/option_detail.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/call"
        android:title="Call"
        android:icon="@drawable/ic_menu_call"
    />
    <item android:id="@+id/photo"
        android:title="Take a Photo"
        android:icon="@drawable/ic_menu_camera"
    />
</menu>
```

We also need to update `onOptionsItemSelected()` in `DetailForm` to watch for this new menu choice, so modify yours to look like:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.call) {
        String toDial="tel:"+phone.getText().toString();

        if (toDial.length()>4) {
            startActivity(new Intent(Intent.ACTION_CALL,
                Uri.parse(toDial)));

            return(true);
        }
    } else if (item.getItemId()==R.id.photo) {
        startActivity(new Intent(this, Photographer.class));

        return(true);
    }

    return(super.onOptionsItemSelected(item));
}
```

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Store the photos on the SD card. Remember: the emulator does not emulate an SD card by default, so you will need to create an SD card image and tell the emulator to mount it.
- Store the photos on the SD card in a background thread, so the user regains control more quickly.
- Associate photos with the current restaurant and be able to view them later (perhaps via a `Gallery`).

Further Reading

More on the `Camera` class for taking still pictures can be found in the "Using the Camera" chapter of [The Busy Coder's Guide to Advanced Android Development](#).

Sensing a Disturbance

Continuing our additions to `LunchList`, we really need a solution to the age-old problem of deciding where to go for lunch. For that, a good old-fashioned random selection would be a fine starting point. But, to make it more entertaining, we should trigger the selection by shaking the device. Shake the phone when on the list of restaurants, and a randomly-selected restaurant will be shown via its `DetailForm`.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 37-Camera edition of `LunchList` to use as a starting point.

Step #1: Implement a Shaker

We need something that hooks into the `SensorManager` and watches for shaking events, where a "shake" is defined as a certain percentage over Earth's gravity, as determined by calculating the total force via the square root of the sum of the squares of all three dimensional forces.

And if that was total gibberish to you, this is why humankind has developed encapsulation and copy-and-paste.

Create `LunchList/src/apt/tutorial/Shaker.java` with the following source:

```
package apt.tutorial;

import android.content.Context;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.os.SystemClock;
import java.util.ArrayList;
import java.util.List;

public class Shaker {
    private SensorManager mgr=null;
    private long lastShakeTimestamp=0;
    private double threshold=1.0d;
    private long gap=0;
    private Shaker.Callback cb=null;

    public Shaker(Context ctxt, double threshold, long gap,
                  Shaker.Callback cb) {
        this.threshold=threshold*threshold;
        this.threshold=this.threshold
                      *SensorManager.GRAVITY_EARTH
                      *SensorManager.GRAVITY_EARTH;
        this.gap=gap;
        this.cb=cb;

        mgr=(SensorManager)ctxt.getSystemService(Context.SENSOR_SERVICE);
        mgr.registerListener(listener,
                             mgr.getDefaultSensor(Sensor.TYPE_ACCELEROMETER),
                             SensorManager.SENSOR_DELAY_UI);
    }

    public void close() {
        mgr.unregisterListener(listener);
    }

    private void isShaking() {
        long now=SystemClock.uptimeMillis();

        if (lastShakeTimestamp==0) {
            lastShakeTimestamp=now;

            if (cb!=null) {
                cb.shakingStarted();
            }
        }
        else {
            lastShakeTimestamp=now;
        }
    }
}
```

```

private void isNotShaking() {
    long now=SystemClock.uptimeMillis();

    if (lastShakeTimestamp>0) {
        if (now-lastShakeTimestamp>gap) {
            lastShakeTimestamp=0;

            if (cb!=null) {
                cb.shakingStopped();
            }
        }
    }
}

public interface Callback {
    void shakingStarted();
    void shakingStopped();
}

private SensorEventListener listener=new SensorEventListener() {
    public void onSensorChanged(SensorEvent e) {
        if (e.sensor.getType()==Sensor.TYPE_ACCELEROMETER) {
            double netForce=e.values[0]*e.values[0];

            netForce+=e.values[1]*e.values[1];
            netForce+=e.values[2]*e.values[2];

            if (threshold<netForce) {
                isShaking();
            }
            else {
                isNotShaking();
            }
        }
    }

    public void onAccuracyChanged(Sensor sensor, int accuracy) {
        // unused
    }
};
}

```

The Shaker class takes four parameters: a Context (used to get the SensorManager), the percentage of Earth's gravity that is considered a "shake", how long the acceleration is below that level before a shaking event is considered over, and a callback object to alert somebody about shaking starting and stopping. The math to figure out if, for a given amount of acceleration, we are shaking, is found in the SensorListener callback object.

As *The Busy Coder's Guide to Advanced Android Development* explains:

The Shaker simply converts the three individual acceleration components into a combined acceleration value (square root of the sum of the squares), then compares that value to Earth's gravity. If the ratio is higher than the supplied threshold, then we consider the device to be presently shaking, and we call the `shakingStarted()` callback method if the device was not shaking before. Once shaking ends, and time elapses, we call `shakingStopped()` on the callback object and assume that the shake has ended. A more robust implementation of `Shaker` would take into account the possibility that the sensor will not be updated for a while after the shake ends, though in reality, normal human movement will ensure that there are some sensor updates, so we can find out when the shaking ends.

Step #2: Hook Into the Shaker

Given that you magically now have a `Shaker` object, we need to tie it into the `LunchList` activity.

First, implement a `Shaker.Callback` object named `onShake` in `LunchList`:

```
private Shaker.Callback onShake=new Shaker.Callback() {
    public void shakingStarted() {
    }

    public void shakingStopped() {
}
};
```

Right now, we do not do anything with the shake events, though we will address that shortcoming in the next section.

Then, add a `Shaker` data member to `LunchList`, called `shaker`.

We do not want to be tying up the accelerometer when `LunchList` does not have the foreground. One way to do this is to override `onResume()` and initialize our Shaker there:

```
@Override  
public void onResume() {  
    super.onResume();  
  
    shaker=new Shaker(this, 1.45d, 500, onShake);  
}
```

...then release our Shaker in an implementation of `onPause()`:

```
@Override  
public void onPause() {  
    super.onPause();  
  
    shaker.close();  
}
```

Now when you shake the device, it will invoke our do-nothing callback.

Step #3: Make a Random Selection on a Shake

To actually choose a restaurant, we need to ask our `RestaurantAdapter` how many restaurants there are, then use `Math.random()` to pick one. We can then package that in an `Intent` and start our `DetailForm` on that restaurant. In particular, we ask our adapter for the `_id` value (`getItemId()`) of the randomly-chosen restaurant, which gets used as the primary key when `DetailForm` looks for it in the database.

With that in mind, here is a revised implementation of the `onShake` callback object:

```
private Shaker.Callback onShake=new Shaker.Callback() {  
    public void shakingStarted() {  
        // no-op - only care when the shaking stops  
    }  
  
    public void shakingStopped() {  
        int selection=(int)(adapter.getCount()*Math.random());  
        Intent i=new Intent(LunchList.this, DetailForm.class);  
    }  
}
```

```
i.putExtra(ID_EXTRA,
    String.valueOf(adapter.getItemId(selection)));
startActivity(i);
};
```

If you rebuild and reinstall the application on a device, you can randomly choose a restaurant by giving the phone a solid shake.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Store the GPS coordinates of the restaurant via the `DetailForm`. Then, given your current position and the location of the restaurant, present a compass to help point you in the right direction, in case you forget where the restaurant is. **WARNING:** this may involve icky math.
- Attempt to use the accelerometer to measure your current speed while walking or jogging. Remind yourself partway through why, exactly, you elected not to be a physics major in college. If you were a physics major in college, the author offers his sincere condolences.

Further Reading

More information about accessing and leveraging the orientation and acceleration sensors in Android is found in the "Sensors" chapter of [The Busy Coder's Guide to Advanced Android Development](#).

Getting the Word Out

If we are getting together with others for lunch, they may not be familiar with every one of the vast array of restaurants that we are tracking in `LunchList`. Hence, it would be nice to be able to send details about the restaurant to people, so that they can find it if they are getting there separately. One likely approach to do this is to send an SMS with the name and address of the restaurant, so that is what we will add to `LunchList` in this tutorial.

This tutorial will only work on Android 2.x, and there is a bug in the Android 2.2 emulator that will make doing this tutorial a bit difficult. Hence, you are best served using Android 2.1 right now. The next tutorial, though, will add backwards compatibility to what we do here, so you can use this feature on earlier versions of Android.

And, of course, to actually send the message, you will need an actual Android phone, not just the emulator.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 38-Sensors edition of `LunchList` to use as a starting point.

Step #1: Add a "Send SMS" Option Menu

The most likely place to add an option to send an SMS about a restaurant would be the `DetailForm` activity, since that is where we have our per-restaurant operations (e.g., call the restaurant). And, since we already have an option menu for `DetailForm`, all we need to do is add another entry to it for sending an SMS:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/call"
        android:title="Call"
        android:icon="@drawable/ic_menu_call"
    />
    <item android:id="@+id/sms"
        android:title="Send SMS"
        android:icon="@drawable/ic_menu_send"
    />
    <item android:id="@+id/photo"
        android:title="Take a Photo"
        android:icon="@drawable/ic_menu_camera"
    />
</menu>
```

Note that you will need a suitable icon, such as `ic_menu_send.png` from the Android SDK.

Then, add a corresponding case to `onOptionsItemSelected()` in `DetailForm`, routing SMS menu requests to an as-yet-unimplemented `sendSMS()` method:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.call) {
        String toDial="tel:"+phone.getText().toString();

        if (toDial.length()>4) {
            startActivity(new Intent(Intent.ACTION_CALL,
                Uri.parse(toDial)));

            return(true);
        }
    }
    else if (item.getItemId()==R.id.photo) {
        startActivity(new Intent(this, Photographer.class));

        return(true);
    }
}
```

```
else if (item.getItemId()==R.id.sms) {  
    sendSMS();  
  
    return(true);  
}  
  
return(super.onOptionsItemSelected(item));  
}
```

Step #2: Find Contacts' Mobile Numbers

To send an SMS, we could just have Android handle the whole thing, by passing control to the user's choice of SMS client application. There, they could pick the contact to send the message to and compose the message, perhaps starting with some prose supplied by `LunchList`.

In this case, though, we are going to send the SMS directly. To do that, we need a phone number of the person to which we should send the message.

The `ContactsContract` content provider in Android 2.x is the way to find out these phone numbers. Specifically, we can query it to find all names and phone numbers, filtering based upon phone type to only get those flagged as mobile numbers.

With that in mind, add this preliminary implementation of the `sendSMS()` method to `DetailForm`:

```
private void sendSMS() {  
    String[] PROJECTION=new String[] { Contacts._ID,  
                                         Contacts.DISPLAY_NAME,  
                                         Phone.NUMBER  
                                    };  
    String[] ARGS={String.valueOf(Phone.TYPE_MOBILE)};  
    final Cursor c=managedQuery(Phone.CONTENT_URI,  
                               PROJECTION, Phone.TYPE+"=?",
                               ARGS, Contacts.DISPLAY_NAME);  
}
```

This gives us a `Cursor`, containing the ID, name, and mobile phone number of everyone in our contacts database, sorted by name. The reason why the `Cursor` is declared as `final` will become apparent later in the tutorial.

Note that you will need to add the `READ_CONTACTS` permission to get this to work, since we are directly accessing the contacts' data. Also, you will need to add imports for `android.provider.ContactsContract.Contacts` and `android.provider.ContactsContract.CommonDataKinds.Phone`.

Step #3: Pick a Person

Now, we need to let the user pick to whom we should send the SMS. Given the Cursor, we can display a `ListView`, or a `Spinner`, or something to let the user make a selection. However, those typically imply another `Activity`, and that seems a bit much for just picking a person by name. Instead, we will use an `AlertDialog`.

So, add the following to the end of `sendSMS()`:

```
new AlertDialog.Builder(this)
    .setTitle("Pick a Person")
    .setCursor(c, onSMSClicked, Contacts.DISPLAY_NAME)
    .show();
```

Here, we pop up an `AlertDialog`, supplying the `Cursor`, indicating that the entries in the `AlertDialog`'s list should be made from the `DISPLAY_NAME` column. We will find out the user's selection, if any, in an `onSMSClicked` listener object that will be defined in the next section.

You will need to add an import to `android.app.AlertDialog` for this to compile.

Step #4: Send the Message

Now, we can add the code to find out the user's choice of recipient and send the SMS message.

First, add the `onSMSClicked` object definition to `sendSMS()`, giving us:

```
private void sendSMS() {
    String[] PROJECTION=new String[] { Contacts._ID,
```

```
        Contacts.DISPLAY_NAME,
        Phone.NUMBER
    );
String[] ARGS={String.valueOf(Phone.TYPE_MOBILE)};
final Cursor c=managedQuery(Phone.CONTENT_URI,
                           PROJECTION, Phone.TYPE+"=?",
                           ARGS, Contacts.DISPLAY_NAME);
DialogInterface.OnClickListener onSMSClicked=
    new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int position) {
        c.moveToPosition(position);

        noReallySendSMS(c.getString(2));
    }
};

new AlertDialog.Builder(this)
    .setTitle("Pick a Person")
    .setCursor(c, onSMSClicked, Contacts.DISPLAY_NAME)
    .show();
}
```

The `onSMSClicked` listener will find out the phone number for the selected person (given the clicked-upon cursor position), then call `noReallySendSMS()` to create and deliver the SMS message.

Then, add `noReallySendSMS()` to `DetailForm` as follows:

```
private void noReallySendSMS(String phone) {
    StringBuilder buf=new StringBuilder("We are going to ");

    buf.append(name.getText());
    buf.append(" at ");
    buf.append(address.getText());
    buf.append(" for lunch!");

    SmsManager
        .getDefaults()
        .sendTextMessage(phone, null, buf.toString(), null, null);
}
```

Here, we build up a message, incorporating the name and address from the `EditText` widgets. Then, we use `SmsManager` to send the message to its desired target.

You will need to add the `SEND_SMS` permission to your manifest and add imports for:

- android.content.DialogInterface
- android.telephony.SmsManager

If you compile and install `LunchList` on a phone and choose a restaurant, you can bring up the option menu:

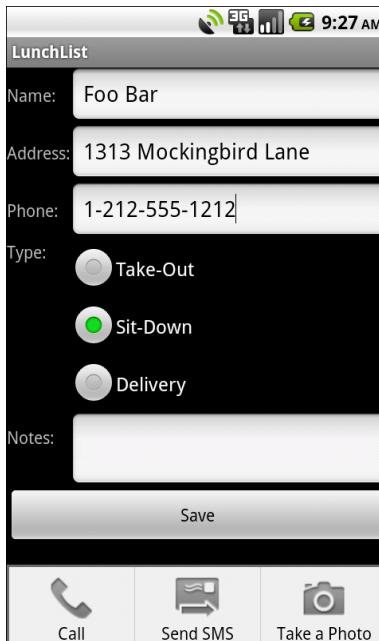


Figure 62. The new option menu on the detail form

Click on the "Send SMS" option, and you will get a dialog with a list of numbers to choose from:

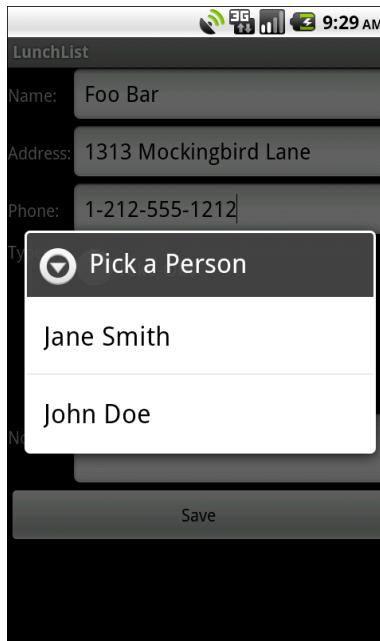


Figure 63. The dialog of contacts with mobile phone numbers

Pick the number, and the SMS is sent.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Also give the user an option to send a restaurant by other means, by using an `ACTION_SEND` Intent and the `createChooser()` method on Intent.
- Experiment with the `PendingIntent` objects you can supply on the call to `sendTextMessage()`, to be notified of the message's progress through the system.

Further Reading

More information about working with SMS in Android is found in the "Working with SMS" chapter of [The Busy Coder's Guide to Advanced Android Development](#).

Seeking the Proper Level

The problem with the preceding tutorial is that it only works on Android 2.x. Specifically, our way of getting the mobile phone numbers will have some issues on previous editions of Android, due to the way the contacts engine was overhauled for Android 2.0.

That being said, we can fix this.

This tutorial will revamp the way we get the mobile numbers, to use separate implementations that work on Android 1.x and Android 2.x. Moreover, we will choose the proper implementation at runtime, ensuring that a 1.x device does not accidentally use the 2.x APIs, or vice versa.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are beginning the tutorials here, or if you wish to not use your existing work, you can [download](#) a ZIP file with all of the tutorial results, and you can copy the 39-SMS edition of `LunchList` to use as a starting point.

Step #1: Define an Interface for Mobile Numbers

We need two pieces of information from the contacts database:

- A cursor containing the contact's ID, name, and mobile phone number
- The name of the field that is the contact's name, so we can supply this to the `AlertDialog.Builder`, so it knows how to display the list

With that in mind, let us define a Java interface, for which we will create implementations using the old and new contacts API. Add the following as `LunchList/src/apt/tutorial/MobileContactsBridge.java`:

```
package apt.tutorial;

import android.app.Activity;
import android.database.Cursor;

interface MobileContactsBridge {
    Cursor getMobileNumbers(Activity host);
    String getDisplayNameField();
}
```

Step #2: Implement the Interface: the New Way

The existing code we added to `DetailForm` in the previous tutorial can be largely copied-and-pasted into a `NewMobileContacts` class that implements this interface. Add the following as `LunchList/src/apt/tutorial/NewMobileContacts.java`:

```
package apt.tutorial;

import android.app.Activity;
import android.database.Cursor;
import android.provider.ContactsContract.Contacts;
import android.provider.ContactsContract.CommonDataKinds.Phone;

class NewMobileContacts implements MobileContactsBridge {
    public Cursor getMobileNumbers(Activity host) {
        String[] PROJECTION=new String[] { Contacts._ID,
                                            Contacts.DISPLAY_NAME,
                                            Phone.NUMBER
                                         };
        String[] ARGS={String.valueOf(Phone.TYPE_MOBILE)};

        return(host.managedQuery(Phone.CONTENT_URI,
                               PROJECTION, Phone.TYPE+"=?",
                               ARGS, Contacts.DISPLAY_NAME));
    }
}
```

```

public String getDisplayNameField() {
    return(Contacts.DISPLAY_NAME);
}

```

We simply generate and return the cursor using the same query as before, and indicate that `Contacts.DISPLAY_NAME` is what should be used from the cursor for the dialog.

Step #3: Implement the Interface: the Old Way

We also need an implementation of this interface that used the old Contacts content provider, as opposed to the new `ContactsContract` content provider. The good news is that while the classes changed, the general concepts are still the same. It is still a content provider, which we can query against to get the data we need. All we need is the proper content Uri, the proper set of projection columns, the right WHERE clause, and so on.

With that in mind, add the following as `LunchList/src/apt/tutorial/OldMobileContacts.java`:

```

package apt.tutorial;

import android.app.Activity;
import android.database.Cursor;
import android.provider.Contacts;

class OldMobileContacts implements MobileContactsBridge {
    public Cursor getMobileNumbers(Activity host) {
        String[] PROJECTION=new String[] { Contacts.Phones._ID,
                                            Contacts.Phones.NAME,
                                            Contacts.Phones.NUMBER
                                         };
        String[] ARGS={String.valueOf(Contacts.Phones.TYPE_MOBILE)};

        return(host.managedQuery(Contacts.Phones.CONTENT_URI,
                               PROJECTION,
                               Contacts.Phones.TYPE+"=?",
                               ARGS,
                               Contacts.Phones.NAME));
    }

    public String getDisplayNameField() {
        return(Contacts.Phones.NAME);
    }
}

```

{}

Step #4: Choose and Use the Bridge

With all that behind us, now we need to actually pick the right implementation of the `MobileContactsBridge` interface and use it, replacing our former hard-wired Android 2.x API usage in `sendSMS()`.

One way to determine which implementation to use is to see what SDK we are running. To do that, we can use the `SDK` property of the `android.os.Build` class (which you will need to add as in `import android.os.Build`). This contains a `String` representation of our API level, with 5 being Android 2.0, 4 being Android 1.6, and so on. Note that while there is an `SDK_INT` property that would save us the `String`-to-integer conversion, it is not available on Android 1.5.

So, replace your current implementation of `sendSMS()` in `DetailForm` to:

```
private void sendSMS() {
    MobileContactsBridge bridge=buildBridge();
    final Cursor c=bridge.getMobileNumbers(this);

    DialogInterface.OnClickListener onSMSClicked=
        new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int position) {
                c.moveToPosition(position);

                noReallySendSMS(c.getString(2));
            }
        };

    new AlertDialog.Builder(this)
        .setTitle("Pick a Person")
        .setCursor(c, onSMSClicked, bridge.getDisplayNameField())
        .show();
}
```

This delegates the creation of the actual `MobileContactsBridge` to a `buildBridge()` method, which you will also need to add to `DetailForm`:

```
private static MobileContactsBridge buildBridge() {
    int sdk=new Integer(Build.VERSION.SDK).intValue();
```

```
if (sdk<5) {
    return(new OldMobileContacts());
}

return(new NewMobileContacts());
```

The `buildBridge()` method looks at the SDK value and chooses an implementation to use. The `sendsSMS()` message takes the cursor and field name from the bridge and pours that information into the `AlertDialog.Builder`.

You can now compile and run `LunchList` on both Android 2.x and Android 1.x.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Rather than determining which API to use by looking at the SDK version, check to see if the `ContactsContract` class exists, since that class does not exist on Android 1.x.
- Experiment with using pure reflection to get at the `Uri` for the content provider based on the version of Android the code is running on.

Further Reading

More information about supporting multiple API levels in Android is found in the "Handling Platform Changes" chapter of [The Busy Coder's Guide to Android Development](#).

How To Get Started

Let us get you set up with the pieces and parts necessary to build an Android app.

If you would prefer not to install all of this software on your development machine, you can [download a Ubuntu VirtualBox image](#) that contains everything described below. Note that this image is 1.6GB and therefore may take some time to download. This image will be updated periodically to reflect newer editions of the Android SDK and development tools. The user account is `android` with a password of `bugdroid`.

NOTE: the instructions presented here are accurate as of the time of this writing. However, the tools change rapidly, and so these instructions may be out of date by the time you read this. Please refer to the [Android Developers Web site](#) for current instructions, using this as a base guideline of what to expect.

Java

When you write Android applications, you typically write them in Java source code. That Java source code is then turned into the stuff that Android actually runs (Dalvik bytecode in an APK file).

Hence, the first thing you need to do is get set up with a Java development environment and be ready to start writing Java classes.

Step #1: Install the JDK

You need to obtain and install the official Sun/Oracle Java SE SDK (JDK). You can obtain this from the [Oracle Java Web site](#) for Windows and Linux, and presumably from Apple for OS X. The plain JDK (sans any "bundles") should suffice. Follow the instructions supplied by Oracle or Apple for installing it on your machine. At the time of this writing, Android supports Java 5 and Java 6, the latter being the now-current edition.

Alternative Java Compilers

In principle, you are supposed to use the official Sun/Oracle Java SE SDK. In practice, it appears that OpenJDK also works, at least on Ubuntu. However, the further removed you get from the official Sun/Oracle implementation, the less likely it is that it will work. For example, the GNU Compiler for Java (GCJ) may not work with Android.

Step #2: Learn Java

This book, like most books and documentation on Android, assumes that you have basic Java programming experience. If you lack this, you really should consider spending a bit of time on Java fundamentals, before you dive into Android. Otherwise, you may find the experience to be frustrating.

If you are in need of a crash course in Java to get involved in Android development, here are the concepts you need to succeed, presented in no particular order:

- Language fundamentals (flow control, etc.)
- Classes and objects
- Methods and data members
- Public, private, and protected
- Static and instance scope

- Exceptions
- Threads and concurrency control
- Collections
- Generics
- File I/O
- Reflection
- Interfaces

Install the Android SDK

The Android SDK gives you all the tools you need to create and test Android applications. It comes in two parts: the base tools, plus version-specific SDKs and related add-ons.

Step #1: Install the Base Tools

The Android developer tools can be found on the [Android Developers Web site](#). Download the ZIP file appropriate for your platform and unZIP it in some likely spot – there is no specific path that is required. Windows users also have the option of running a self-installing EXE file.

Step #2: Install the SDKs and Add-Ons

Inside the `tools/` directory of your Android SDK installation from the previous step, you will see an `android` batch file or shell script. If you run that, you will be presented with the Android SDK and AVD Manager:



Figure 64. Android SDK and AVD Manager

At this point, while you have some of the build tools, you lack the Java files necessary to compile an Android application. You also lack a few additional build tools, plus the files necessary to run an Android emulator.

To address this, click on the Available Packages option on the left. This brings up a tree:

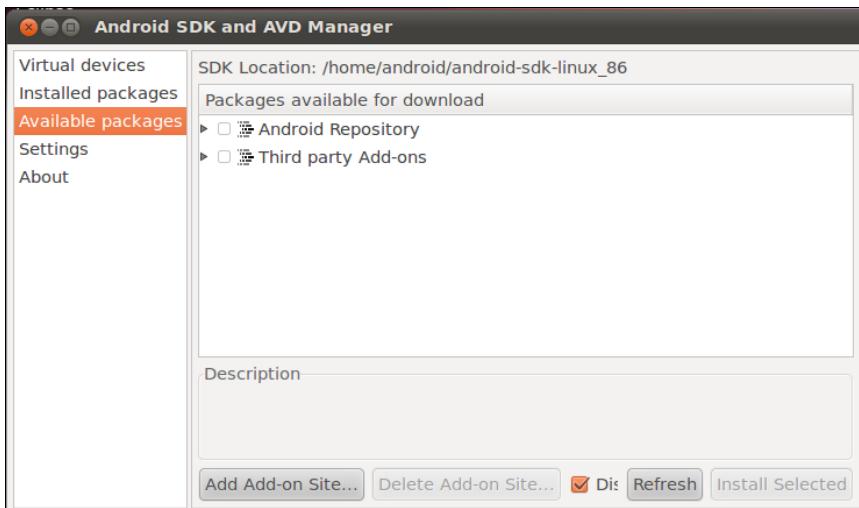


Figure 65. Android SDK and AVD Manager Available Packages

Open the Android Repository branch of the tree. After a short pause, you will see a screen similar to this:

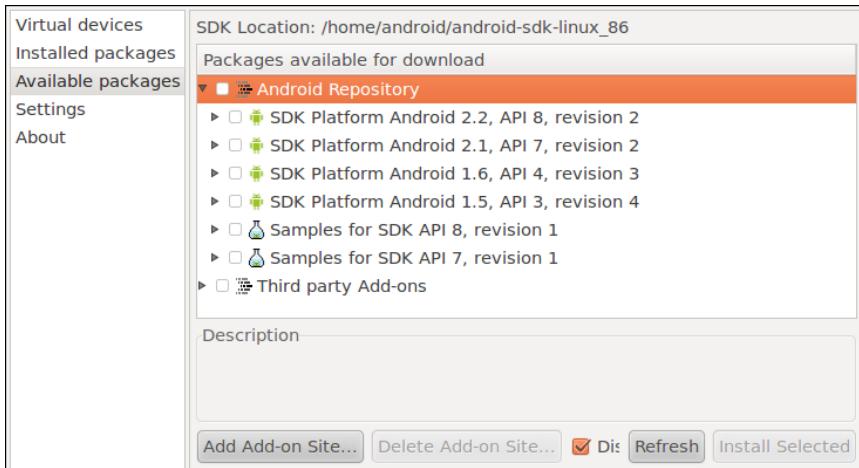


Figure 66. Android SDK and AVD Manager Available Android Packages

You will want to check the following items:

- "SDK Platform" for all Android SDK releases you want to test against
- "Documentation for Android SDK" for the latest Android SDK release
- "Samples for SDK" for the latest Android SDK release, and perhaps for older releases if you wish

Then, open the Third-Party Add-Ons branch of the tree. After a short pause, you will see a screen similar to this:

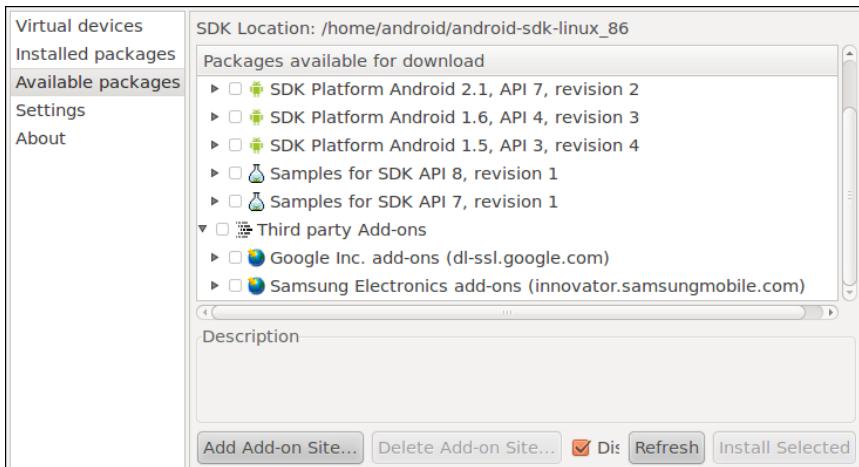


Figure 67. Android SDK and AVD Manager Available Third-Party Add-Ons

Fold open the "Google Inc. add-ons" branch, which will display something like this:

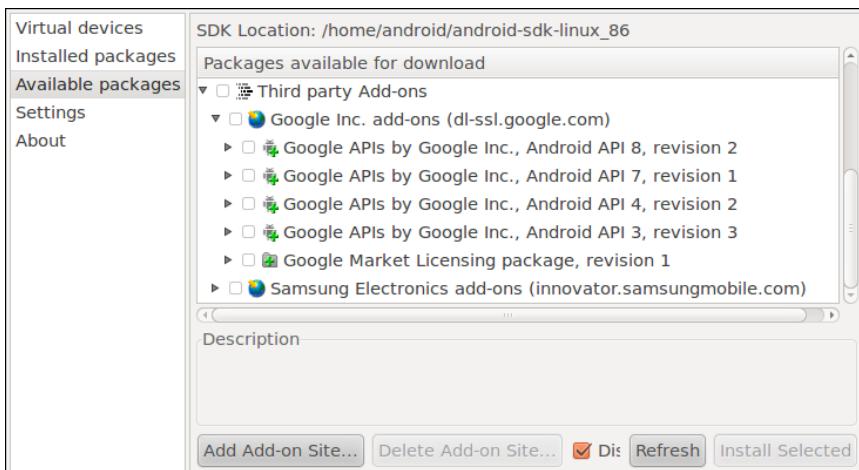


Figure 68. Android SDK and AVD Manager Available Google Add-Ons

Most likely, you will want to check the "Google APIs by Google Inc." items that match up with the SDK versions you selected in the Android Repository branch. The "Google APIs" include support for Google Maps, both from your code and in the Android emulator.

When you have checked all of the items you want to download, click the Install Selected button, which brings up a license confirmation dialog:

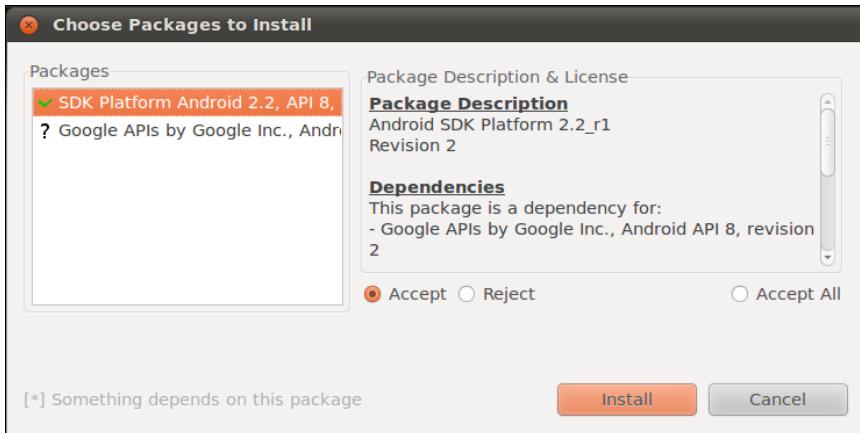


Figure 69. Android SDK and AVD Manager Installing Packages

Review and accept the licenses, then click the Install button. At this point, this is a fine time to go get lunch. Or, perhaps dinner. Unless you have a substantial Internet connection, downloading all of this data and unpacking it will take a fair bit of time.

When the download is complete, you can close up the SDK and AVD Manager if you wish, though we will use it to set up the emulator in [a later step of this chapter](#).

Install the ADT for Eclipse

If you will not be using Eclipse for your Android development, you can skip to the next section.

If you have not yet installed Eclipse, you will need to do that first. Eclipse can be downloaded from the [Eclipse Web site](#). The "Eclipse IDE for Java Developers" package will work fine.

Next, you need to install the Android Developer Tools (ADT) plug-in. To do this, go to Help | Install New Software... in the Eclipse main menu. Then, click the Add button to add a new source of plug-ins. Give it some name (e.g., Android) and supply the following URL: <https://dl-ssl.google.com/android/eclipse/>. That should trigger Eclipse to download the roster of plug-ins available from that site:

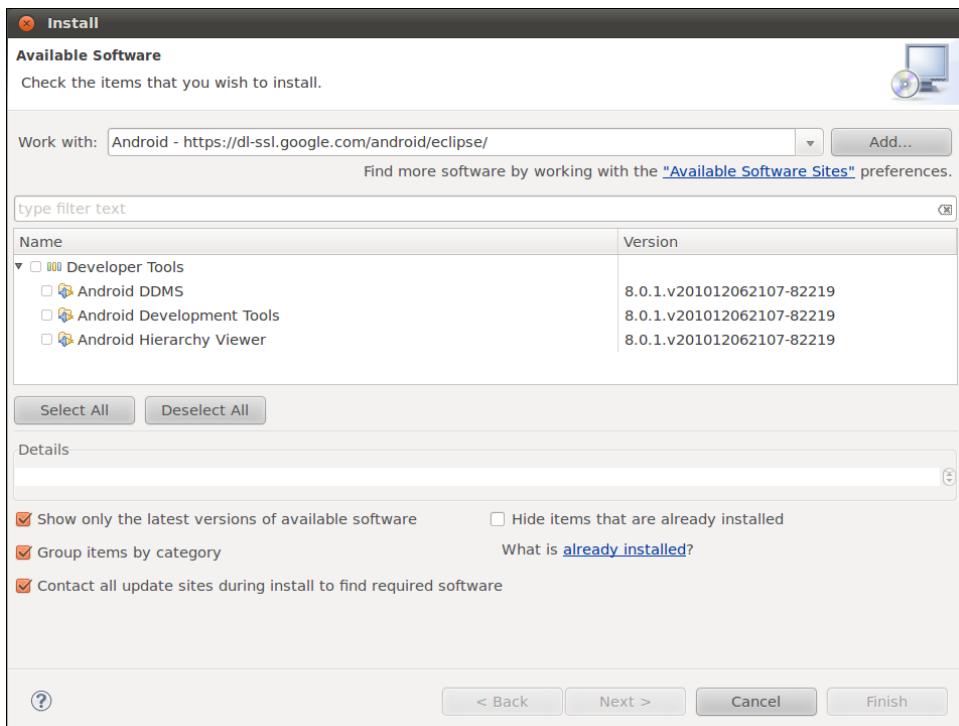


Figure 70. Eclipse ADT plug-in installation

Check the checkbox to the left of "Developer Tools" and click the Next button. Follow the rest of the wizard to review the tools to be downloaded and their respective license agreements. When the Finish button is enabled, click it, and Eclipse will download and install the plug-ins. When done, Eclipse will ask to restart – please let it.

Then, you need to teach ADT where your Android SDK installation is from [the preceding section](#). To do this, choose Window | Preferences from the

Eclipse main menu (or the equivalent Preferences option for OS X). Click on the Android entry in the list on the left:

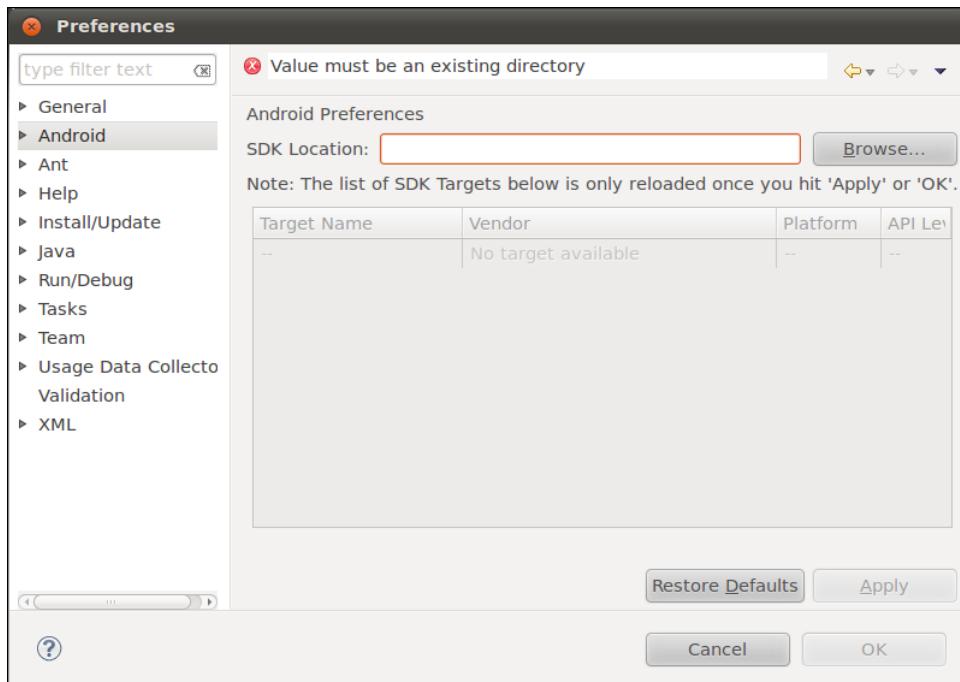


Figure 71. Eclipse ADT configuration

Then, click the Browse... button to find the directory where you installed the SDK. After choosing it, click Apply on the Preferences window, and you should see the Android SDK versions you installed previously. Then, click OK, and the ADT will be ready for use.

You may also wish to read the appendix with tips on [working with the tutorials in Eclipse](#).

Install Apache Ant

If you will be doing all of your development from Eclipse, you can skip to the next section.

If you wish to develop using command-line build tools, you will need to install Apache Ant. You may have this already from previous Java development work, as it is fairly common in Java projects. However, you will need Ant version 1.8.1, so double-check your current copy (e.g., `ant -version`) to ensure you are on the proper edition.

If you do not have Ant, you can obtain it from the [Apache Ant Web site](#). They have [full installation instructions](#) in the Ant manual, but the basic steps are:

1. Unpack the ZIP archive wherever it may make sense on your machine
2. Add a `JAVA_HOME` environment variable, pointing to where your JDK is installed, if you do not have one already
3. Add an `ANT_HOME` environment variable, pointing to the directory where you unpacked Ant in the first step above
4. Add `$JAVA_HOME/bin` and `$ANT_HOME/bin` to your PATH
5. Run `ant -version` to confirm that Ant is installed properly

Set Up the Emulator

The Android tools include an emulator, a piece of software that pretends to be an Android device. This is very useful for development – not only does it mean you can get started on Android without a device, but the emulator can help test device configurations that you do not own.

The Android emulator can emulate one or several Android devices. Each configuration you want is stored in an "Android Virtual Device", or AVD. The SDK and AVD Manager, which you used to download the SDK components [earlier in this chapter](#), is where you create these AVDs.

If you do not have the SDK and AVD Manager running, you can run it via the `android` command from your SDK's `tools/` directory, or via Window | SDK and AVD Manager from Eclipse. It starts up on a screen listing the AVDs you have available – initially, the list will be empty:



Figure 72. Android SDK and AVD Manager

Click the New... button to create a new AVD file. This brings up a dialog where you can configure what this AVD should look and work like:

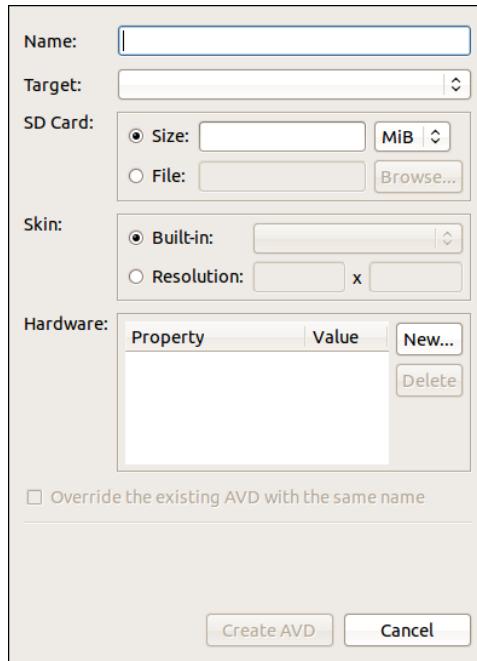


Figure 73. Adding a New AVD

You need to provide the following:

- A name for the AVD. Since the name goes into files on your development machine, you will be limited by filename conventions for your operating system (e.g., no backslashes on Windows).
- The Android version you want the emulator to run (a.k.a., the "target"). Choose one of the SDKs you installed via the drop-down list. Note that in addition to "pure" Android environments, you will have options based on the third-party add-ons you selected. For example, you probably have some options for setting up AVDs containing the Google APIs, and you will need such an AVD for testing an application that uses Google Maps.
- Details about the SD card the emulator should emulate. Since Android devices invariably have some form of "external storage", you probably want to set up an SD card, by supplying a size in the associated field. However, since a file will be created on your development machine of whatever size you specify for the card, you probably do not want to create a 2GB emulated SD card. 32MB is a nice starting point, though you can go larger if needed.
- The "skin" or resolution the emulator should run in. The skin options you have will depend upon what target you chose. The skins let you choose a typical Android screen resolution (e.g., WVGA800 for 800x480). You can also manually specify a resolution when you want to test a non-standard configuration.

You can skip the "Hardware" section for now, as changing those settings is usually only required for advanced configurations.

The resulting dialog might look something like this:

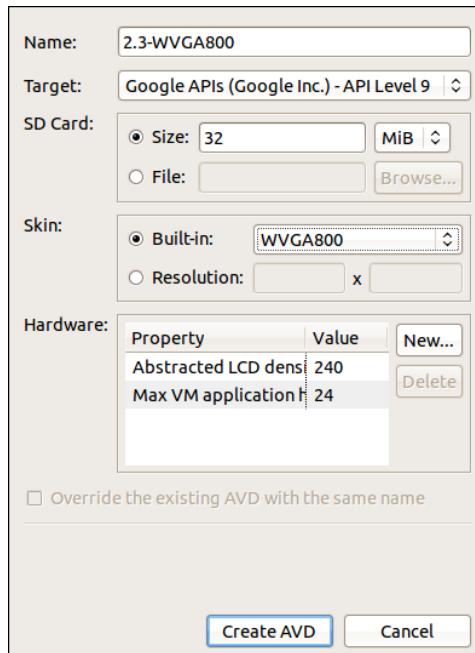


Figure 74. Adding a New AVD (continued)

Click the Create AVD button, and your AVD stub will be created.

To start the emulator, highlight it in the list and click Start... You can skip the launch options for now and just click Launch. The first time you launch a new AVD, it will take a long time to start up. The second and subsequent times you start the AVD, it will come up a bit faster, and usually you only need to start it up once per day (e.g., when you start development). You do not need to stop and restart the emulator every time you want to test your application, in most cases.

The emulator will go through a few startup phases, first with a plain-text "ANDROID" label:

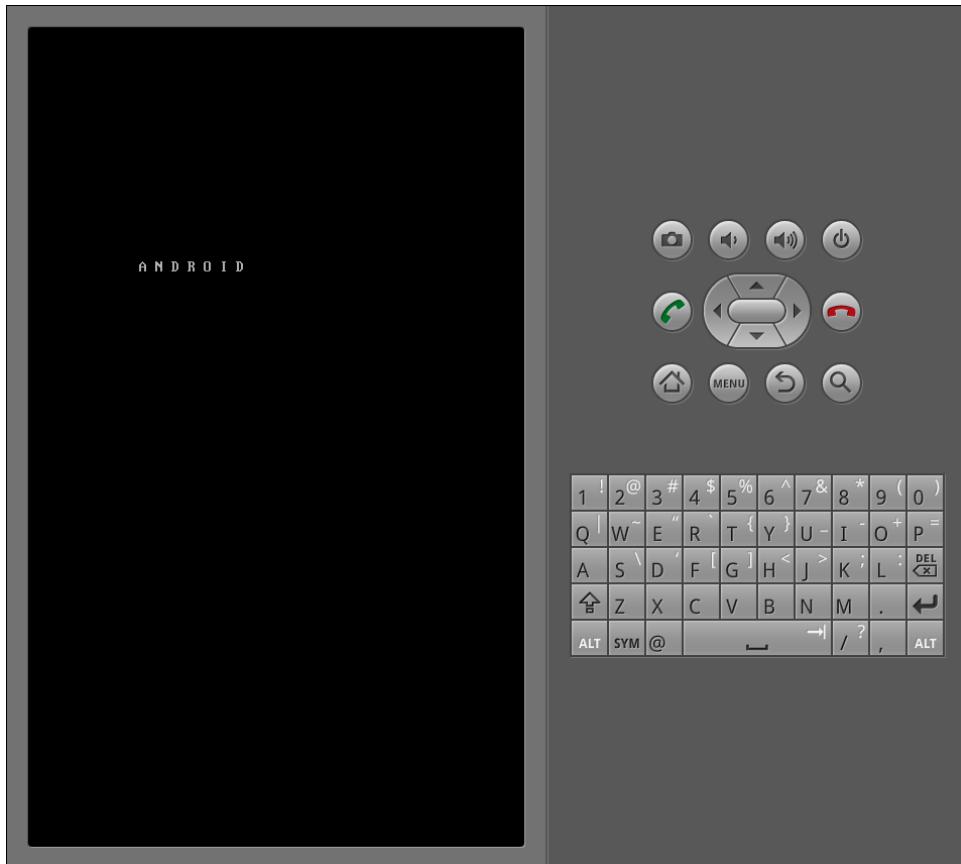


Figure 75. Android emulator, initial startup segment

...then a graphical Android logo:

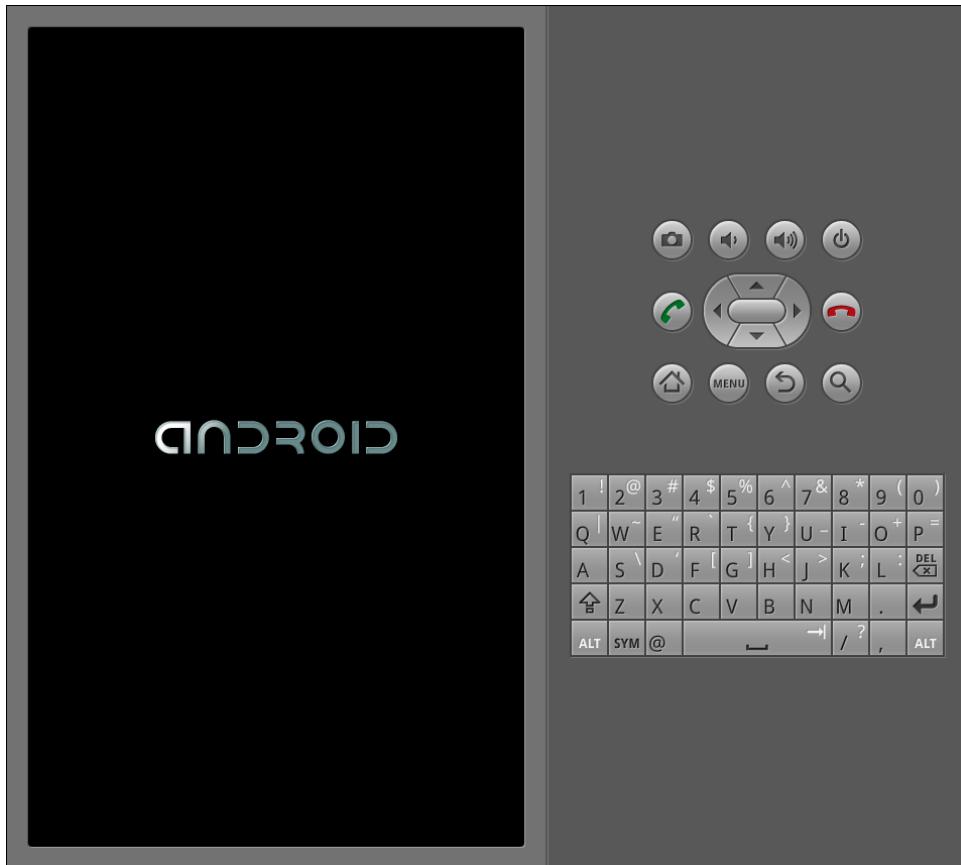


Figure 76. Android emulator, secondary startup segment

before eventually landing at the home screen (the first time you run the AVD, shown below) or the keyguard:

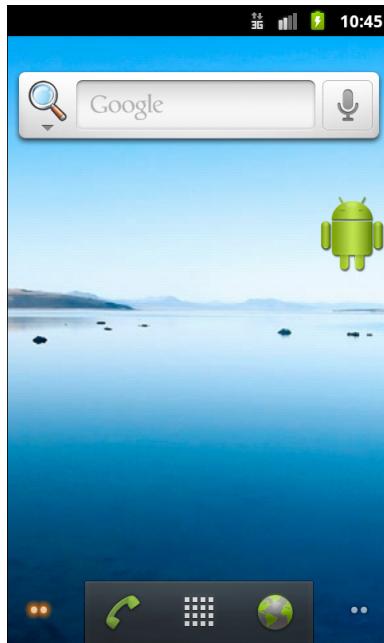


Figure 77. Android home screen

If you get the keyguard (shown below), press the MENU button, or slide the green lock on the screen to the right, to get to the emulator's home screen:



Figure 78. Android keyguard

Set Up the Device

You do not need an Android device to get started in Android application development. Having one is a good idea before you try to ship an application (e.g., upload it to the Android Market). And, perhaps you already have a device – maybe that is what is spurring your interest in developing for Android.

The first step to make your device ready for use with development is to go into the *Settings* application on the device. From there, choose *Applications*, then *Development*. That should give you a set of checkboxes of development-related options to consider:

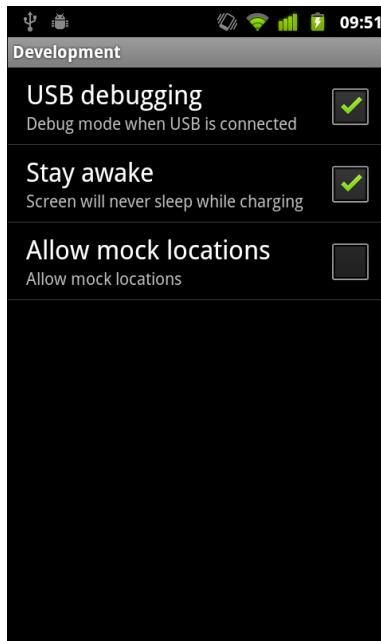


Figure 79. Android device development settings

Generally, you will want to enable USB debugging, so you can use your device with the Android build tools. You can leave the other settings alone for now if you wish, though you may find the "Stay awake" option to be handy, as it saves you from having to unlock your phone all of the time while it is plugged into USB.

Next, you need to get your development machine set up to talk to your device. That process varies by the operating system of your development machine, as is covered in the following sections.

Step #1: Windows

When you first plug in your Android device, Windows will attempt to find a driver for it. It is possible that, by virtue of other software you have installed, that the driver is ready for use. If it finds a driver, you are probably ready to go.

If the driver is not found, here are some options for getting one.

Windows Update

Some versions of Windows (e.g., Vista) will prompt you to search Windows Update for drivers. This is certainly worth a shot, though not every device will have supplied its driver to Microsoft.

Standard Android Driver

In your Android SDK installation, you will find a `google-usb_driver` directory, containing a generic Windows driver for Android devices. You can try pointing the driver wizard at this directory to see if it thinks this driver is suitable for your device.

Manufacturer-Supplied Driver

If you still do not have a driver, search the CD that came with the device (if any) or search the Web site of the device manufacturer. [Motorola](#), for example, has drivers available for all of their devices in one spot for download.

Step #2: OS X and Linux

Odds are decent that simply plugging in your device will "just work". You can see if Android recognizes your device via running `adb devices` in a shell (e.g., OS X Terminal), where `adb` is in your `platform-tools/` directory of your SDK. If you get output similar to the following, Android detected your device:

```
List of devices attached
HT9CPP809576    device
```

If you are running Ubuntu (or perhaps other Linux variants), and this command did not work, you may need to add some `udev` rules. For example,

here is a `51-android.rules` file that will handle the devices from a handful of manufacturers:

```
SUBSYSTEM=="usb", SYSFS{idVendor}=="0bb4", MODE="0666"
SUBSYSTEM=="usb", SYSFS{idVendor}=="22b8", MODE="0666"
SUBSYSTEM=="usb", SYSFS{idVendor}=="18d1", MODE="0666"
SUBSYSTEMS=="usb", ATTRS{idVendor}=="18d1", ATTRS{idProduct}=="0c01",
MODE="0666", OWNER="[me]"
SUBSYSTEM=="usb", SYSFS{idVendor}=="19d2", SYSFS{idProduct}=="1354", MODE="0666"
SUBSYSTEM=="usb", SYSFS{idVendor}=="04e8", SYSFS{idProduct}=="681c", MODE="0666"
```

Drop that in your `/etc/udev/rules.d` directory on Ubuntu, then either reboot the computer or otherwise reload the udev rules (e.g., `sudo service udev reload`). Then, unplug and re-plug in the device and see if it is detected.

Coping with Eclipse

The author of this book is not an Eclipse user, which is why this book aims to be agnostic in terms of development tools, unlike many other Android resources that depict Eclipse as being mandatory.

That being said, Eclipse is a fine tool for Android development, but not everything may be necessarily obvious. If you elect to use Eclipse, here are some tips for getting around some of the Android aspects. Note that these tips are workarounds cobbled together from assisting developers in live Android training sessions, and so there may be better solutions available from true Eclipse experts.

How to Import a Non-Eclipse Project

Not all Android projects ship with Eclipse project files, such as the sample projects associated with this book. However, these can still be easily added to your Eclipse workspace, if you wish. Here is how to do it!

First, choose File > New > Project... from the Eclipse main menu:

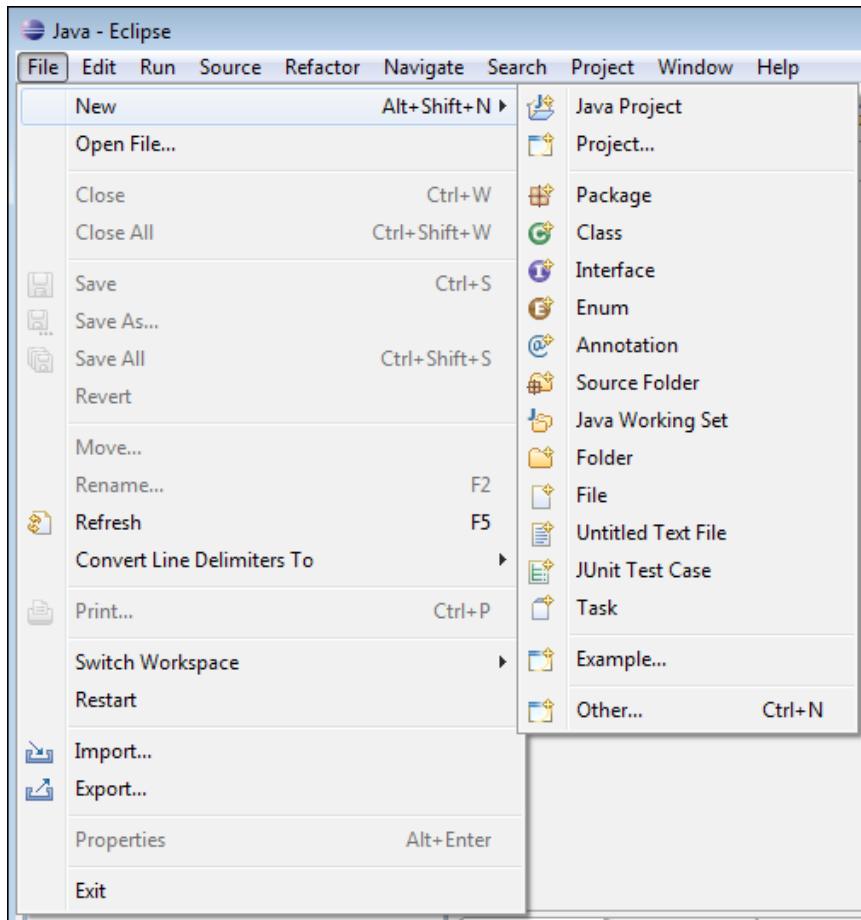


Figure 80. File menu in Eclipse

Then, choose **Android > Android Project** from the tree of available project types:

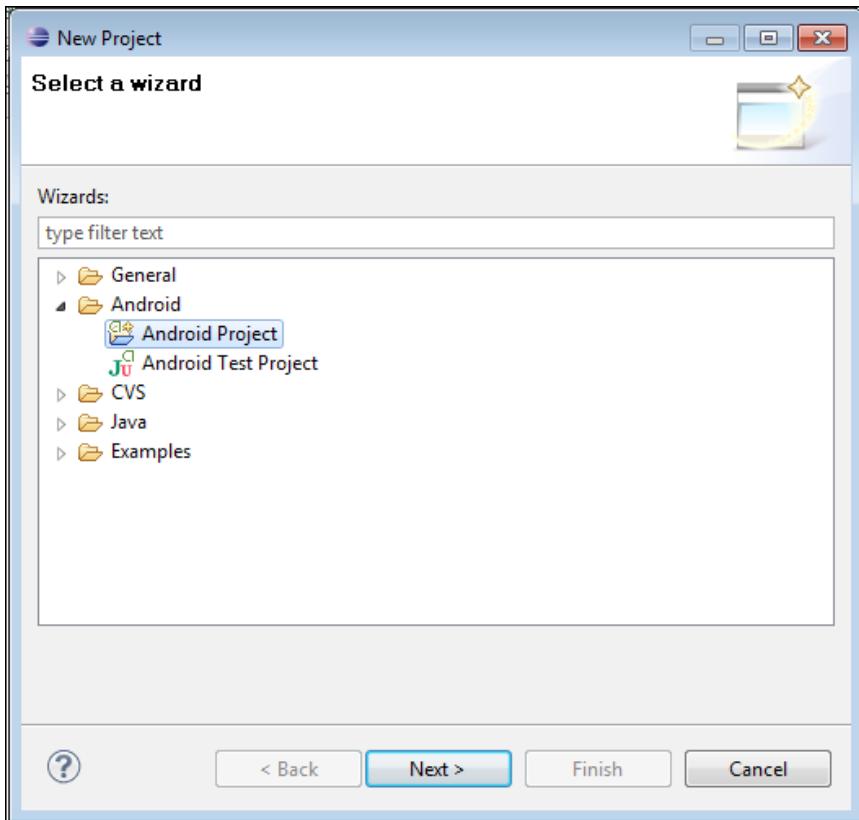


Figure 81. New project wizard in Eclipse

Note: if you do not see this option, you have not installed the Android Developer Tools.

Then, in the next page of the project creation wizard, choose the "Create project from existing source" radio button, click the [Browse...] button, and open the directory containing your project's `AndroidManifest.xml` file. This will populate most of the rest of this screen, though you may need to also specify a build target from the table:

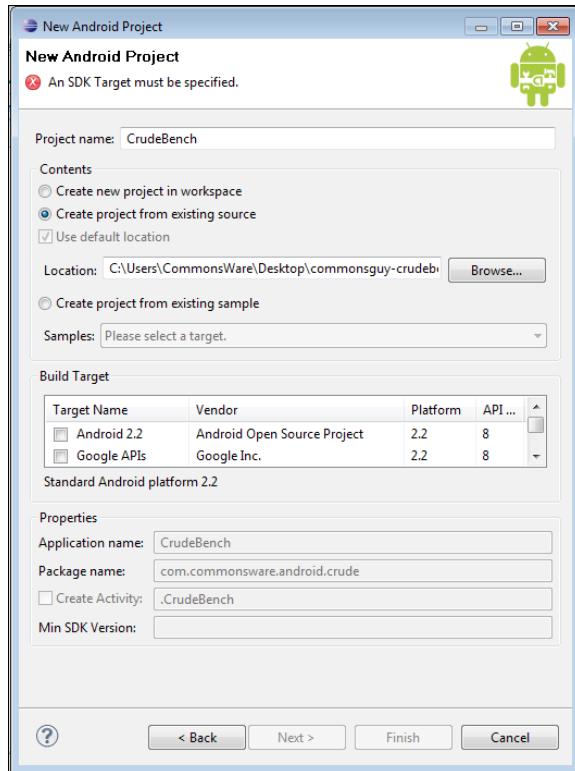


Figure 82. Android project wizard in Eclipse

Then, click [Finish]. This will return you to Eclipse, with the imported project in your workspace:

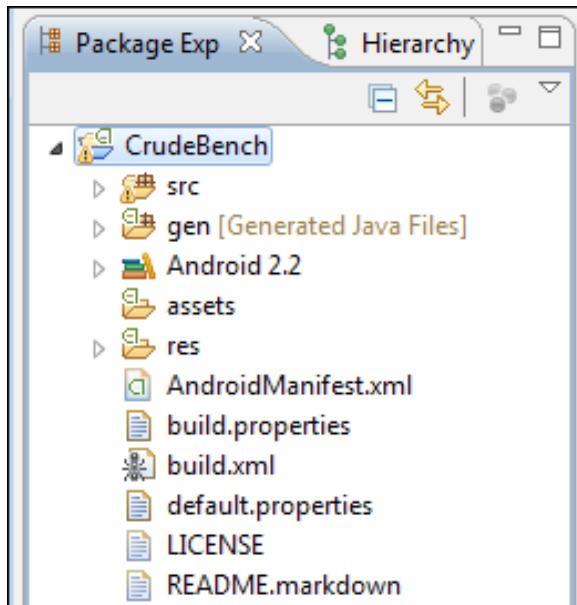


Figure 83. Android project tree in Eclipse

Next, right-click over the project name, and choose Build Path > Configure Build Path from the context menu:

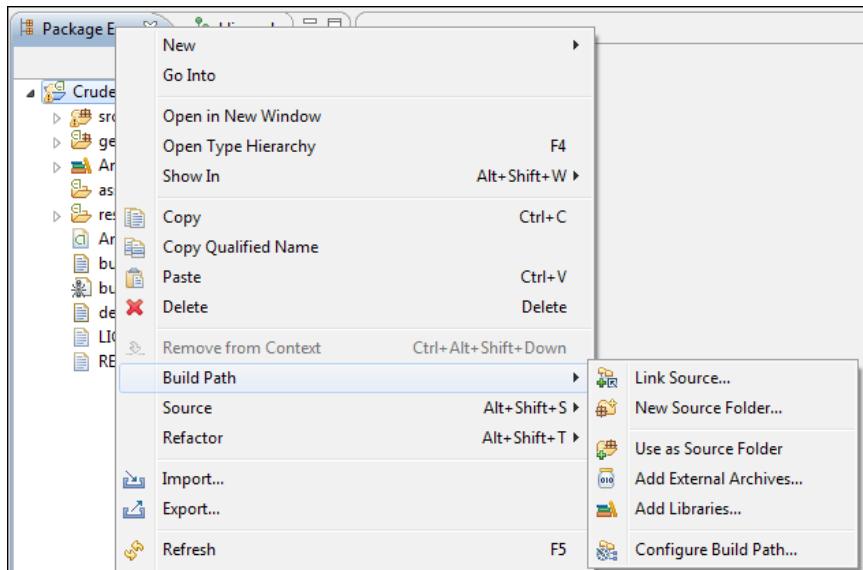


Figure 84. Project context menu in Eclipse

This brings up the build path portion of the project properties window:

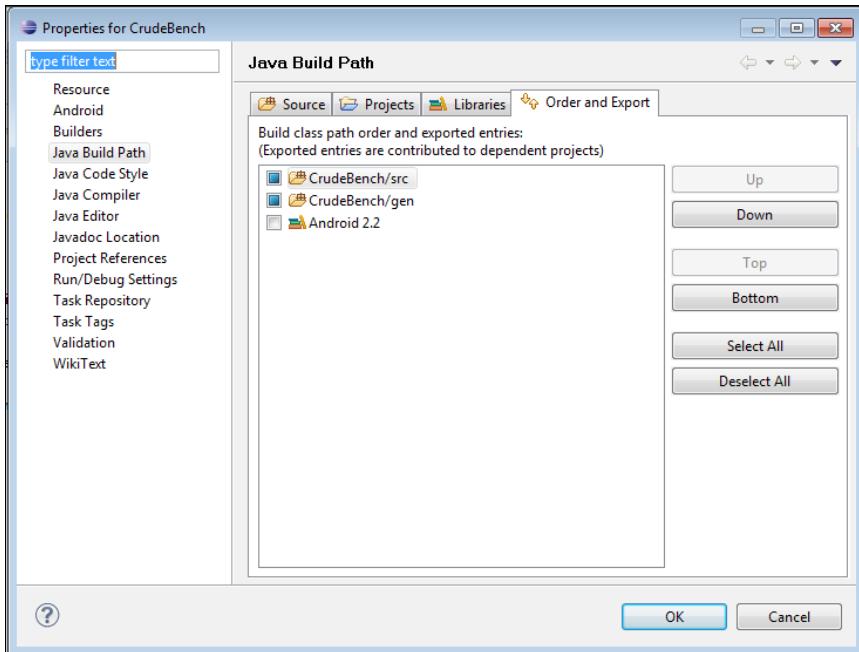


Figure 85. Project properties window in Eclipse

If the Android JAR is not checked (see the Android 2.2 entry in the above image), check it, then close the properties window. At this point, your project should be ready for use.

How to Get To DDMS

Many times, you will be told to take a look at something in DDMS, such as the LogCat tab to examine Java stack traces. In Eclipse, DDMS is a perspective. To open this perspective in your workspace, choose Window > Open Perspective > Other... from the main menu:

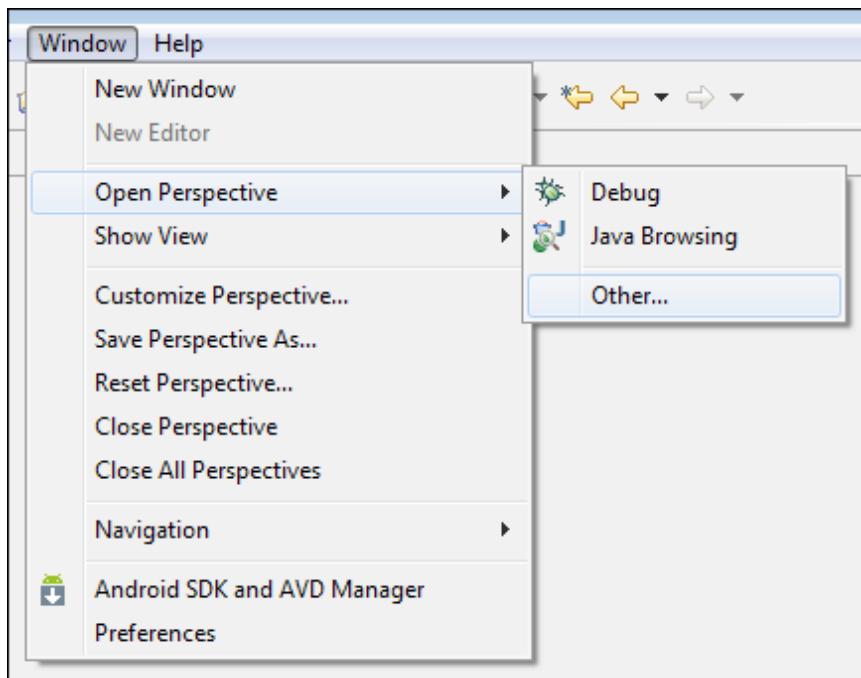


Figure 86. Perspective menu in Eclipse

Then, in the list of perspectives, choose DDMS:

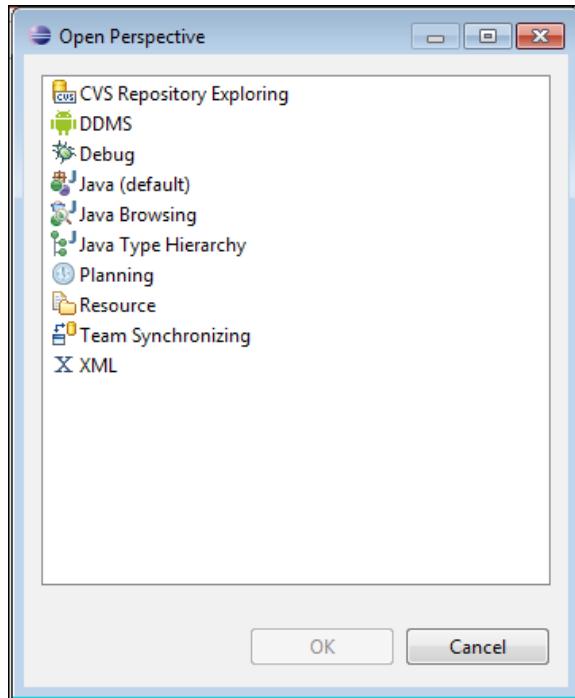


Figure 87. Perspective roster in Eclipse

This will add the DDMS perspective to your workspace and open it in your Eclipse IDE.

How to Create an Emulator

By default, your Eclipse environment has no Android emulators set up. You will need one before you can run your project successfully.

To do this, first choose Window > Android SDK and AVD Manager from the main menu:

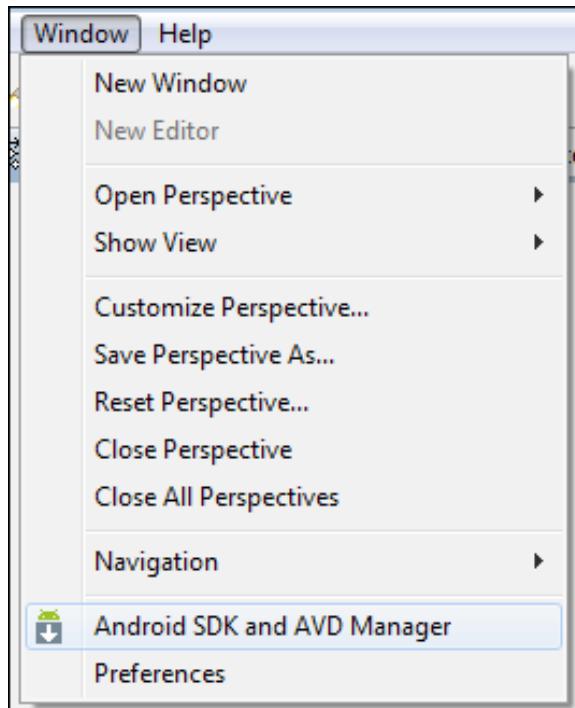


Figure 88. Android AVD Manager menu option in Eclipse

That brings up the same window as you can get by running `android` from the command line.

How to Run a Project

Given that you have an AVD defined, or that you have a device set up for debugging and connected to your development machine, you can run your project in the emulator.

First, click the Run toolbar button, or choose Project > Run from the main menu. This will bring up the "Run As" dialog the first time you run the project:

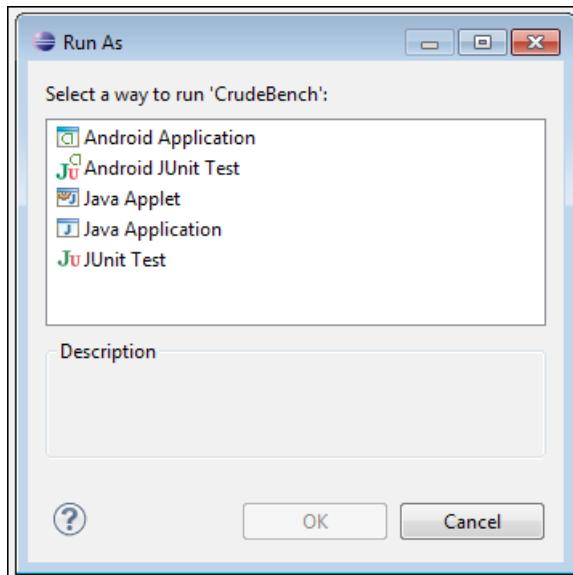


Figure 89. Android AVD Manager menu option in Eclipse

Choose Android Application and click OK. If you have more than one AVD or device available, you will be presented with a window where you choose the desired target environment. Then, the emulator will start up to run your application. Note that you will need to unlock the lock screen on the emulator (or device) if it is locked.

How Not to Run Your Project

When you go to run your project, be sure to not have an XML file be the active tab in the editor. Attempting to "run" this will result in a .out file being created in whatever directory the XML file lives in (e.g., res/layout/main.xml.out). To recover, simply delete the offending .out file and try running again, this time with a Java file as the active tab.

How to Get Past Eclipse

Some people will tell you that the only sensible way to do Android development is to use Eclipse. These people are sorely mistaken.

Eclipse – whether with the basic Android add-on or the full-blown **MOTODEV Studio for Android** – is a fine Java IDE. It is not the only Java IDE. And, many developers are able to get by quite nicely without an IDE.

If you are used to using Eclipse, stick with it.

If you are used to working outside of Eclipse, you are welcome to give that a try and perhaps continue to work outside of Eclipse. There is nothing in Android development that can *only* be done with Eclipse. IntelliJ's IDEA, for example, has Android support as an integrated add-on that ships with their product, so if you are using IDEA, there is no particular reason to change. Or, you are welcome to avoid an IDE altogether, as many developers do, including the author of this book.

Do not be afraid to try Eclipse, but at the same time, do not feel compelled to use Eclipse.

Keyword Index

Class.....	
Account.....	184, 185
ActionAdapter.....	309, 310
Activity.....	15, 16, 38, 49, 69, 86, 220, 306, 366
ActivityInfo.....	312
Adapter.....	45
AdapterView.OnItemClickListener.....	51
AlarmManager.....	289, 290, 292, 293, 298
AlertDialog.....	67, 133, 161, 163, 366
AlertDialog.Builder.....	372, 375
AlphaAnimation.....	263, 270
AndroidHttpClient.....	163
Animation.....	269
AnimationListener.....	269
AnimationSet.....	270
AnimationUtils.....	269
AppWidget.....	317, 321, 329, 330
AppWidgetManager.....	329
AppWidgetProvider.....	316, 317, 327
ArrayAdapter.....	33, 93, 99
ArrayList.....	33, 52, 96, 99, 101, 190
AsyncTask.....	76, 200
AtomicBoolean.....	78, 178, 287
AutoCompleteTextView.....	36
Base64.....	159
BroadcastListener.....	285
BroadcastReceiver.....	286, 290, 316
Bundle.....	147, 148
Button.....	16, 30, 56, 254, 308
Camera.....	353, 355
Contacter.....	305-307, 309, 310
ContactsContract.....	283, 365, 375
ContentValues.....	97
Context.....	329, 359
Criteria.....	232
Cursor.....	96, 98-102, 123, 143, 282, 365, 366, 372, 373, 375
CursorAdapter.....	93, 99, 102, 200
CustomHelp.....	258, 259
DatePicker.....	56

Keyword Index

DatePickerDialog.....	56	IPostMonitor.Stub.....	208
DefaultHttpClient.....	161, 163	KillJob.....	76
DetailForm	109-117, 119, 122, 124, 129, 131, 147-149, 157, 331-333, 342, 344, 345, 354, 357, 361, 362, 364, 365, 367, 372, 374	LinearLayout.....	17, 20, 39, 90, 264
Dialer.....	345	LinkedBlockingQueue.....	75
EditPreference.....	168, 169	List.....	99
EditPreferences.....	137-139	ListActivity.....	114, 116, 306
EditText	15-17, 20, 36, 45, 57, 59, 90, 161, 163, 168, 231, 342, 367	ListAdapter.....	37
EditTextPreference.....	168	ListView.....	29, 31, 33, 36, 37, 47, 51, 114, 188, 306, 308, 310, 316, 366
ErrorDialog.....	67	LocalBinder.....	181, 187, 200, 208
Exception.....	161	Location.....	231
FakeJob.....	75	LocationListener.....	229
FrameLayout.....	47, 49	LocationManager.....	227, 228
Gallery.....	355	LocationProvider.....	232
Handler.....	75	LunchList....xxvi-29, 33, 38, 41, 42, 49, 51, 53, 58-60, 62, 63, 69, 71, 72, 78, 81, 87, 88, 93, 95-97, 99-102, 110, 111, 113-115, 117-119, 123, 126, 129, 138, 142, 155, 157, 168, 169, 189, 299, 300, 302, 304, 305, 315, 316, 323, 327, 335, 336, 349, 357, 360, 363, 365, 368, 375	
HelpCast.....	247, 253, 254	Map.....	274
HelpcastActivity.....	249	MapActivity.....	236
HelpPage.....	253-255, 257, 258	MediaController.....	247
HttpPost.....	161	Menu.....	170, 345
ImageButton.....	325, 326, 333	MenuItem.....	170, 345
ImageView.....	41	MobileContactsBridge.....	374
Intent....xxviii, 111, 115, 117, 170, 220, 221, 271-274, 301, 307-309, 332, 345, 346, 361, 369		MyLocationOverlay.....	243
IntentFilter.....	286	NewMobileContacts.....	372
Intents.....	291, 293	Notification.....	219, 220, 223, 225, 298
IntentService.....	292, 293, 327, 329	NotificationManager.....	220
IPostListener.....	184, 187, 205, 206	OnAlarmReceiver.....	291
IPostListener.Stub.....	206	OnClickListener.....	330
IPostMonitor.....	184, 187		

Keyword Index

PackageManager.....	305, 309, 313	Shaker.Callback.....	360
Patchy...xxvii, 28, 155, 156, 158, 159, 166, 167, 169, 172, 180, 181, 184, 186-190, 192, 193, 200, 204-209, 219-221, 223-225, 228, 229, 231, 236-238, 240, 245, 247, 251-253, 260, 264-267, 269-271, 274, 275, 280-283, 285, 288, 289, 294, 298, 305		SharedPreferences.....	136, 142, 171
Pattern.....	238	SmsManager.....	367
PendingIntent.....	220, 292, 293, 330-333, 369	Spinner.....	36, 174, 249, 366
PhotoCallback.....	354	SQLiteDatabase.....	97, 98
Photographer.....	351, 352, 354	SQLiteOpenHelper.....	93
PostMonitor. 177, 180, 181, 184, 186, 187, 198, 200, 204-209, 214, 219-221, 225, 271-274, 285-291, 294, 298		StatusEntryView.....	265
Preference.....	225	StatusMap.....	237-240, 252, 256
PreferenceActivity.....	135, 136	StatusOverlay.....	240
PreferenceScreen.....	135, 136	String.....	57, 116, 294, 303, 374
ProgressBar.....	69	SurfaceView.....	351-354
RadioButton.....	21, 22, 24, 26	TabActivity.....	49, 114
RadioGroup.....	22, 26	TabHost.....	47, 49
RelativeLayout.....	31, 47	TableLayout.....	19, 20, 22, 27, 58, 157, 264, 342
RemoteViews.....	322, 330	TableRow.....	20, 58, 170
ResolveInfo.....	309, 310, 312	TabView.....	47
RestaurantAdapter....	38, 40, 42, 97, 99, 100, 114, 361	TabWidget.....	47
RestaurantHelper	94-96, 98, 101, 105, 116, 117, 119, 131, 141, 143, 301, 339, 340	TextView.....	17, 41, 251, 322, 324
RestaurantHolder.....	37, 42, 100	Thread.....	75, 76, 78
Runnable.....	71, 74, 77, 78, 95, 178, 292, 293	TimelineAdapter.....	191, 192
ScrollView.....	26	TimelineEntry. .189, 190, 192, 238, 256, 274, 280-282	
SensorListener.....	359	TimelineEntryWrapper.....	190, 281
SensorManager.....	357, 359	Toast.....	57, 61, 62, 67, 162, 242, 316, 333
ServiceConnection.....	186, 206	tools/.....	386
Set.....	180	Twitter.....	167, 170-172, 175, 179, 180
Shaker.....	359-361	Uri.....	307, 345, 375
		VideoView.....	246, 247
		View.....	99, 264
		View.OnClickListener.....	16
		ViewFlipper.....	56

Keyword Index

WakefulIntentService.....289-293, 298

WakeLock.....289-291, 293

WebView.....256-260

WebViewClient.....255

WidgetService.....327, 329-331

Command.....

adb devices.....395

adb logcat.....21, 336

adb pull.....107

adb push.....245

adb shell.....107

android.....379, 405

android create project.....156, 204, 305

android list targets.....5

android update project -pxxi

ant -version.....386

ant clean.....21

ant install.....16, 21

ant jar.....290

ant reinstall.....26

pdftk *.pdf cat output combined.pdf.....xviii

sqlite3.....107

sudo service udev reload.....396

telnet localhost 5554.....288

Method.....

add().....33

addIntentOptions().....312

addView().....26

bindService().....186

bindView().....99

buildBridge().....374, 375

clearNotification().....223, 224

createChooser().....369

doSomeLongWork().....71, 74, 95

doWakefulWork().....293

findViewById().....16

finish().....119

getActivity().....332

getAll().....100, 141-143, 301, 341

getById().....341

getCheckedRadioButtonId().....26

getClient().....172

getFollowers().....174

getFriendsTimeline().....180

getItemId().....361

getItemViewType().....45

getStatus().....175

getSystemService().....220

getTag().....42

getView().....42, 99

getViewTypeCount().....45

getWritableDatabase().....97

initList().....143, 144, 300

insert().....96, 97, 102, 340

insertLocation().....231

load().....117

managedQuery().....282

newView().....99

noReallySendSMS().....367

Keyword Index

onActivityResult().....	309	registerReceiver().....	285
onBind().....	182	removeAccount().....	207
onClick().....	31, 51	requestWindowFeature().....	95
onCreate()..	38, 49, 51, 59, 69, 75, 86, 92, 94-96, 100, 112-114, 116, 117, 142, 143, 148, 171, 179, 186, 187, 191, 223, 224, 228, 229, 237, 240, 257, 258, 265, 269, 285, 292, 294, 298, 307, 339, 340, 353	resetClient().....	187, 207
onCreateOptionsMenu().....	118	runOnUiThread().....	74
onCreateOptionsMenu().....	95	save().....	119
onDestroy()..	96, 116, 179, 187, 207, 229, 286, 294	sendBroadcast().....	273
onHandleIntent().....	329, 331	sendMessage().....	174
onItemClick().....	52	sendSMS().....	364-366, 374, 375
onKeyDown().....	232	sendTextMessage().....	369
onListItemClick().....	116	sendWakefulWork().....	291
onNewIntent().....	224	setAlarm().....	292, 293
onOptionsItemSelected()..	62, 72, 73, 95, 118, 138, 231, 247, 253, 267, 300, 354, 364	setContentView().....	51, 69, 96, 110, 112
onPause().....	78, 79, 95, 361	setListAdapter().....	114
onPrepareOptionsMenu().....	265	setOnItemClickListener().....	51
onRestoreInstanceState().....	148, 343	setTag().....	42
onResume().....	78, 79, 95, 144, 360	shakingStarted().....	360
onRetainNonConfigurationInstance().....	150	shakingStopped().....	360
onSaveInstanceState()..	86, 92, 147, 148, 150, 343	shouldOverrideUrlLoading().....	255
onSearchRequested().....	300	showNotification().....	220, 221
onStart().....	86	start().....	247
onStop().....	86	startActivity().....	111, 133, 220, 238, 254
onUpdate().....	316, 321, 327, 329, 330	startActivityForResult().....	308
onUpgrade().....	94, 95, 340	startManagingCursor().....	100
poll().....	180, 185, 200, 209, 221, 225, 272	startService().....	291, 330
populateFrom().....	281	startWork().....	79, 95
rawQuery().....	98	stopManagingCursor().....	143
registerAccount().....	200	surfaceChanged().....	353
		toggleStatusEntry().....	267, 269
		toString().....	31

Keyword Index

unregisterReceiver().....	286	updateStatus().....	161, 167, 172
update().....	119, 341		