Lauren Darcey
Shane Conder

Third Edition
Features Android 4.0

Android™ Wireless Application Development

Volume II: Advanced Topics

Developer's Library

Android™ Wireless Application Development

Volume II: Advanced Topics

Third Edition

This page intentionally left blank

Android™ Wireless Application Development

Volume II: Advanced Topics

Third Edition

Lauren Darcey
Shane Conder

Addison-Wesley

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

Library of Congress Cataloging-in-Publication Data is on file.

ISBN-13: 978-0-321-81384-8
ISBN-10: 0-321-81384-7

Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.
First printing, July 2012

Editor-in-Chief
Mark Taub

Acquisitions Editor
Laura Lewin

Development Editor
Songlin Qiu

Managing Editor
Kristy Hart

Project Editor
Betsy Harris

Copy Editor
Deadline-Driven Publishing

Indexer
Lisa Stumpf

Proofreader
Paula Lowell

Technical Reviewers
Tony Hillerson
Douglas Jones
Ray Rischpater

Publishing Coordinator
Olivia Basegio

Multimedia Developer
Dan Scherf

Book Designer
Gary Adair

Senior Compositor
Gloria Schurick

❖

This book is dedicated to ESC.

❖

This page intentionally left blank

Contents

V: Maximizing Android's Unique Features

Acknowledgments

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

About the Authors

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

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

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

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

The authors have also published several other Android books, including *Android Wireless Application Development*, *Android Wireless Application Development Volume I: Android Essentials*, *Sams Teach Yourself Android Application Development*, *Learning Android™ Application Programming for the Kindle Fire™*, and the mini-book *Introducing Android Development with Ice Cream Sandwich*. Lauren and Shane have also published numerous articles on mobile software development for magazines, technical journals, and online

publishers of educational content. You can find dozens of samples of their work in *Linux User and Developer*, *Smart Developer* magazine (Linux New Media), developer.com, Network World, Envato (MobileTuts+ and CodeCanyon), and InformIT, among others. They also publish articles of interest to their readers at their own Android website, http://androidbook.blogspot.com. You can find a full list of the authors' publications at http://goo.gl/f0Vlj.

Introduction

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

Who Should Read This Book?

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

This book was written for several audiences:

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

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

Why Two Volumes in the Third Edition?

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

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

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

Android Wireless Application Development Volume II: Advanced Topics is divided into seven parts. Here is an overview of the various parts in this book:

- **Part I: Advanced Android Application Design Principles**

  Part I picks up where *Android Wireless Application Development Volume I: Android Essentials* leaves off in terms of application design techniques. We begin by talking about asynchronous processing. We then move on to some of the more complex Android application components, such as services, application databases (SQLite), content providers, and intents and notifications.

- **Part II: Advanced Android User Interface Design Principles**

  Part II dives deeper into some of the more advanced user interface tools and techniques available as part of the Android SDK, including working with action bars and menus, gathering input through nonstandard methods such as gestures and

voice recognition, and much more. You also learn more about how to develop applications that are accessible to different types of users with impairments.

- **Part III: Leveraging Common Android APIs**

  Part III dives deeper into some of the more advanced and specialty APIs available as part of the Android SDK, including networking, location-based services, multimedia (including the camera), telephony, and hardware sensors.

- **Part IV: Drawing, Animations, and Graphics Programming with Android**

  Part IV is for those developers incorporating graphics of any kind into their applications. We cover both 2D and 3D graphics (OpenGL ES and RenderScript), animation, and the Android NDK.

- **Part V: Maximizing Android's Unique Features**

  Part V discusses some of the many ways the Android platform is different from other mobile platforms and how your applications can leverage its unique features. Here you learn how to extend your application features beyond the traditional borders of mobile applications, integrating them with the Android operating system. App Widgets, enabling searches, leveraging cloud-based services, and backups are just some of the topics discussed.

- **Part VI: Advanced Topics in Application Publication and Distribution**

  Part VI covers some more specialized topics in the realm of application publication and distribution, including how to internationalize your applications, enable In-App billing with the Android Market, track application usage patterns with Google Analytics, and take measures to protect your intellectual property from software pirates.

- **Part VII: Appendixes**

  Part VII includes a helpful quick start guide for the Android Debug Bridge tool and a refresher course on using SQLite.

Key Questions Answered in Volume II

This volume of the book answers the following questions:

1. How can developers write responsive applications?
2. How are Android applications structured? How are background operations handled with services? What are broadcast intents and how can applications use them effectively?
3. How do applications store data persistently using SQLite? How can applications act as content providers and why would they want to do so?
4. How do applications interact with the Android operating system? How do applications trigger system notifications, access underlying device hardware, and monitor device sensors?

5. How can developers design the best user interfaces for the devices of today and tomorrow? How can developers work with 2D and 3D graphics and leverage animation opportunities on Android?
6. How can developers write high-performance, computationally intensive applications using native code or RenderScript?
7. What are some of the most commonly used APIs for networking, location-based services, multimedia, telephony, and Internet access?
8. What do managers, developers, and testers need to look for when planning, developing, and testing a mobile development application?
9. How do mobile teams design bulletproof Android applications for publication?
10. How can developers make their applications leverage everything Android has to offer in the form of App Widgets, live wallpapers, and other system perks?
11. How can applications take advantage of some of the optional third-party APIs available for use, such as the Android Market's In-App billing and license verification libraries, Google's Analytics, and Cloud-to-Device Messaging (C2DM) services?

An Overview of Changes in This Edition

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

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

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

- Coverage of the latest and greatest Android tools and utilities.
- Updates to all existing chapters, often with entirely new sections.
- New chapters, which cover new SDK features or expand upon those covered in previous editions.
- Updated sample code and applications, conveniently organized by chapter.
- Topics such as threading and asynchronous processing, creating content providers, broadcast intents, and animation frameworks now have their own chapters.
- Coverage of hot topics such as tablet and TV design, best practices, Renderscript, in-app billing, and Google Analytics.
- Even more tips and tricks from the trenches to help you design, develop, and test applications for different device targets, including an all-new chapter on tackling compatibility issues.

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

The Development Environment Used in This Book

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

Windows 7 and Mac OS X 10.7.x

- Eclipse Java IDE Version 3.7 (Indigo)
- Eclipse JDT plug-in and Web Tools Platform (WTP)
- Java SE Development Kit (JDK) 6 Update 26
- Android SDK Version 2.3.4, API Level 10 (Gingerbread MR1), Android SDK Version 3.2, API Level 13 (Honeycomb MR2), Android SDK Version 4.0.3, API Level 15 (Ice Cream Sandwich MR1)
    1. ADT plug-in for Eclipse 16.0.1
    2. SDK Tools Revision 16
    3. Android Support Package r4
    4. Android NDK r7
- Android devices: Samsung Galaxy Nexus, Motorola Droid 3, Samsung Galaxy tab 10.1, Asus Transformer Prime, Motorola Atrix 4G, and Logitech Revue

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

Android's latest major platform update, Android 4.0, frequently called by its code-name, Ice Cream Sandwich or just ICS, merges the smartphone-centric Android 2.3.x (Gingerbread) and the tablet-centric Android 3.x (Honeycomb) platform editions into a single SDK for all smart-devices, be they phones, tablets, televisions, or toasters. This book features the latest SDK and tools available, but it does not focus on them to the detriment of popular legacy versions of the platform. This book is meant to be an overall reference to help developers support all popular devices on the market today. As of the writing of this book, only a small percentage (less than 5 percent) of users' devices run Android 3.0 or 4.0. Of course, some devices receive upgrades, and users purchase new devices as they become available, but for now, developers need to straddle this gap and support numerous versions of Android to reach the majority of users in the field.

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

Supplementary Materials Available

The source code that accompanies this book is available for download on the publisher website: http://www.informit.com/title/9780321813848. The source code is also available for download from our book website: http://androidbook.blogspot.com/p/book-code-downloads.html (http://goo.gl/kyAsN). You can also find a variety of Android topics discussed at our book website (http://androidbook.blogspot.com). For example, we present reader feedback, questions, and additional information. You can also find links to our various technical articles on our book website.

Where to Find More Information

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

- **Android Developer Website:** The Android SDK and developer reference site: http://developer.android.com/

- **Stack Overflow:** The Android website with great technical information (complete with tags) and an official support forum for developers:
  http://stackoverflow.com/questions/tagged/android
- **Open Handset Alliance:** Android manufacturers, operators, and developers:
  http://www.openhandsetalliance.com/
- **Android Market:** Buy and sell Android applications:
  http://www.android.com/market/
- **Mobiletuts+:** Mobile development tutorials, including Android:
  http://mobile.tutsplus.com/category/tutorials/android/
- **anddev.org:** An Android developer forum:
  http://www.anddev.org
- **Google Team Android Apps:** Open source Android applications:
  http://apps-for-android.googlecode.com/
- **Android Tools Project Site:** The tools team discusses updates and changes:
  https://sites.google.com/a/android.com/tools/recent
- **FierceDeveloper:** A weekly newsletter for wireless developers:
  http://www.fiercedeveloper.com/
- **Wireless Developer Network:** Daily news on the wireless industry:
  http://www.wirelessdevnet.com/
- **XDA-Developers Android Forum**: From general development to ROMs:
  http://forum.xda-developers.com/forumdisplay.php?f=564
- **Developer.com:** A developer-oriented site with mobile articles:
  http://www.developer.com/

Conventions Used in This Book

This book uses the following conventions:

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

This book also presents information in the following sidebars:

Tip

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

Note

Notes provide additional information that might be interesting or relevant.

Warning

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

Contacting the Authors

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

http://androidbook.blogspot.com
Or, email us at:

androidwirelessdev+awad3ev2@gmail.com
Circle us on Google+:

- Lauren Darcey: http://goo.gl/P3RGo
- Shane Conder: http://goo.gl/BpVJh

3

Leveraging SQLite Application Databases

Applications use a combination of application preferences, the file system, and database support to store information. In this chapter, we explore one of the most powerful ways you can store, manage, and share application data with Android: an application database powered by SQLite. Application databases provide structured data storage that is quick to access, search, and manipulate.

Note

For more information about designing SQLite databases and interacting with them via the `sqlite3` command-line tool, please see Appendix B, "The SQLite Quick-Start Guide." This appendix is divided into two parts. The first half is an overview of the most commonly used features of the `sqlite3` command-line interface and the limitations of SQLite compared to other flavors of SQL; the second half of the appendix includes a fully functional tutorial in which you build a SQLite database from the ground up and then use it. If you are new to SQLite or a bit rusty on your syntax, this appendix is for you.

Storing Structured Data Using SQLite Databases

When your application requires a more robust data storage mechanism, you'll be happy to hear that the Android file system includes support for application-specific relational databases using SQLite. SQLite databases are lightweight and file-based, making them ideally suited for embedded devices.

Tip

Many of the code examples provided in this section are taken from the SimpleDatabase application. This source code for the SimpleDatabase application is provided for download on the book's websites.

These databases and the data in them are private to the application. To share application data with other applications, you must expose the data you want to share by making your application a content provider.

The Android SDK includes a number of useful SQLite database management classes. Many of these classes are found in the `android.database.sqlite` package. Here you can find utility classes for managing database creation and versioning, database management, and query builder helper classes to help you format proper SQL statements and queries. The package also includes specialized `Cursor` objects for iterating query results. You can also find all the specialized exceptions associated with SQLite.

In this chapter, we focus on creating databases in our Android applications. For that, we use the built-in SQLite support to programmatically create and use a SQLite database to store application information. However, if your application works with a different sort of database, you can also find more generic database classes (in the `android.database` package) to help you work with data from other providers.

In addition to programmatically creating and using SQLite databases, developers can also interact directly with their application's database using the `sqlite3` command-line tool that's accessible through the ADB shell interface. This can be a helpful debugging tool for developers and quality assurance personnel who might want to manage the database state (and content) for testing purposes.

Creating a SQLite Database

You can create a SQLite database for your Android application in several ways. To illustrate how to create and use a simple SQLite database, let's create an Android project called SimpleDatabase.

Creating a SQLite Database Instance Using the Application Context

The simplest way to create a new `SQLiteDatabase` instance for your application is to use the `openOrCreateDatabase()` method of your application `Context`, like this:

```
import android.database.sqlite.SQLiteDatabase;
...
SQLiteDatabase mDatabase;
mDatabase = openOrCreateDatabase(
    "my_sqlite_database.db",
    SQLiteDatabase.CREATE_IF_NECESSARY,
    null);
```

Finding the Application Database File on the Device File System

Android applications store their databases (SQLite or otherwise) in a special application directory:

```
/data/data/<application package name>/databases/<databasename>
```

So, in this case, the path to the database would be

```
/data/data/com.androidbook.SimpleDatabase/databases/my_sqlite_database.db
```

You can access your database using the `sqlite3` command-line interface using this path.

Configuring the SQLite Database Properties

Now that you have a valid `SQLiteDatabase` instance, it's time to configure it. Some important database configuration options include version, locale, and the thread-safe locking feature:

```
import java.util.Locale;
...
mDatabase.setLocale(Locale.getDefault());
mDatabase.setLockingEnabled(true);
mDatabase.setVersion(1);
```

Creating Tables and Other SQLite Schema Objects

Creating tables and other SQLite schema objects is as simple as forming proper SQLite statements and executing them. The following is a valid `CREATE TABLE` SQL statement. This statement creates a table called `tbl_authors`. The table has three fields: a unique `id` number, which auto-increments with each record and acts as our primary key, and `firstname` and `lastname` text fields:

```
CREATE TABLE tbl_authors (
id INTEGER PRIMARY KEY AUTOINCREMENT,
firstname TEXT,
lastname TEXT);
```

You can encapsulate this `CREATE TABLE` SQL statement in a `static final String` variable (called `CREATE_AUTHOR_TABLE`) and then execute it on your database using the `execSQL()` method:

```
mDatabase.execSQL(CREATE_AUTHOR_TABLE);
```

The `execSQL()` method works for nonqueries. You can use it to execute any valid SQLite SQL statement. For example, you can use it to create, update, and delete tables, views, triggers, and other common SQL objects. In our application, we add another table called `tbl_books`. The schema for `tbl_books` looks like this:

```
CREATE TABLE tbl_books (
id INTEGER PRIMARY KEY AUTOINCREMENT,
title TEXT,
dateadded DATE,
authorid INTEGER NOT NULL CONSTRAINT authorid REFERENCES tbl_authors(id) ON DELETE
CASCADE);
```

Unfortunately, SQLite does not enforce foreign key constraints. Instead, we must enforce them ourselves using custom SQL triggers. So we create triggers, such as this one that enforces that books have valid authors:

```
private static final String CREATE_TRIGGER_ADD =
"CREATE TRIGGER fk_insert_book BEFORE INSERT ON tbl_books
FOR EACH ROW
BEGIN
SELECT RAISE(ROLLBACK, 'insert on table \"tbl_books\" violates foreign key
constraint \"fk_authorid\"') WHERE  (SELECT id FROM tbl_authors WHERE id =
NEW.authorid) IS NULL;
END;";
```

We can then create the trigger simply by executing the `CREATE TRIGGER` SQL statement:

```
mDatabase.execSQL(CREATE_TRIGGER_ADD);
```

We need to add several more triggers to help enforce our link between the author and book tables, one for updating `tbl_books` and one for deleting records from `tbl_authors`.

Creating, Updating, and Deleting Database Records

Now that we have a database set up, we need to create some data. The `SQLiteDatabase` class includes three convenience methods to do that. They are, as you might expect, `insert()`, `update()`, and `delete()`.

Inserting Records

We use the `insert()` method to add new data to our tables. We use the `ContentValues` object to pair the column names to the column values for the record we want to insert. For example, here we insert a record into `tbl_authors` for J.K. Rowling:

```
import android.content.ContentValues;
...
ContentValues values = new ContentValues();
values.put("firstname", "J.K.");
values.put("lastname", "Rowling");
long newAuthorID = mDatabase.insert("tbl_authors", null, values);
```

The `insert()` method returns the identifier of the newly created record. We use this author identifier to create book records for this author.

Tip

There is also another helpful method called `insertOrThrow()`, which does the same thing as the `insert()` method but throws a `SQLException` on failure, which can be helpful, especially if your inserts do not seem to be working and you'd like to know why. Generally, you'll want to check values before inserting and not rely on exceptions for common constraints.

You might want to create simple classes (that is, class `Author` and class `Book`) to encapsulate your application record data when it is used programmatically.

Updating Records

You can modify records in the database using the `update()` method. The `update()` method takes four arguments:

- The table to update records
- A `ContentValues` object with the modified fields to update
- An optional `WHERE` clause, in which `?` identifies a `WHERE` clause argument
- An array of `WHERE` clause arguments, each of which is substituted in place of the `?`s from the second parameter

Passing `null` to the `WHERE` clause modifies all records within the table, which can be useful for making sweeping changes to your database.

Most of the time, we want to modify individual records by their unique identifier. The following function takes two parameters: an updated book title and a `bookId`. We find the record in the table called `tbl_books` that corresponds with the `id` and update that book's title. Again, we use the `ContentValues` object to bind our column names to our data values:

```
public void updateBookTitle(Integer bookId, String newtitle) {
    ContentValues values = new ContentValues();
    values.put("title", newtitle);
    mDatabase.update("tbl_books",
        values, "id=?", new String[] { bookId.toString() });
}
```

Because we are not updating the other fields, we do not need to include them in the `ContentValues` object. We include only the `title` field because it is the only field we change.

Deleting Records

You can remove records from the database using the `remove()` method. The `remove()` method takes three arguments:

- The table to delete the record from
- An optional `WHERE` clause, in which `?` identifies a `WHERE` clause argument

- An array of `WHERE` clause arguments, each of which is substituted in place of the `?`s from the second parameter

Passing `null` to the `WHERE` clause deletes all records in the table. For example, this function call deletes all records in the table called `tbl_authors`:

```
mDatabase.delete("tbl_authors", null, null);
```

Most of the time, though, we want to delete individual records by their unique identifiers. The following function takes a parameter `bookId` and deletes the record corresponding to that unique `id` (primary key) in the table called `tbl_books`:

```
public void deleteBook(Integer bookId) {
    mDatabase.delete("tbl_books", "id=?",
        new String[] { bookId.toString() });
}
```

You need not use the primary key (`id`) to delete records; the `WHERE` clause is entirely up to you. For instance, the following function deletes all book records in the table `tbl_books` for a given author by the author's unique identifier:

```
public void deleteBooksByAuthor(Integer authorID) {
    int numBooksDeleted = mDatabase.delete("tbl_books", "authorid=?",
        new String[] { authorID.toString() });
}
```

Working with Transactions

Often you have multiple database operations you want to happen all together or not at all. You can use SQL transactions to group operations together; if any of the operations fails, you can handle the error and either recover or roll back all operations. If the operations all succeed, you can then commit them. Here we have the basic structure for a transaction:

```
mDatabase.beginTransaction();
try {
    // Insert some records, update others, delete a few.
    // Do whatever you need to do as a unit, then commit it.

    mDatabase.setTransactionSuccessful();
} catch (Exception e) {
    // Transaction failed. Failed! Do something here.
    // It's up to you.
} finally {
    mDatabase.endTransaction();
}
```

Now let's look at the transaction in a bit more detail. A transaction always begins with a call to `beginTransaction()` method and a `try/catch` block. If your operations are successful, you can commit your changes with a call to the `setTransactionSuccessful()` method. If you do not call this method, all your operations are rolled back and not committed. Finally, you end your transaction by calling `endTransaction()` in the finally clause, guaranteeing that it'll be called. It's as simple as that.

In some cases, you might recover from an exception and continue with the transaction. For example, if you have an exception for a read-only database, you can open the database and retry your operations.

Finally, note that transactions can be nested, with the outer transaction either committing or rolling back all inner transactions.

Querying SQLite Databases

Databases are great for storing data in any number of ways, but retrieving the data you want is what makes databases powerful. This is partly a matter of designing an appropriate database schema and partly achieved by crafting SQL queries, most of which are `SELECT` statements.

Android provides many ways in which you can query your application database. You can run raw SQL query statements (strings), use a number of different SQL statement builder utility classes to generate proper query statements from the ground up, and bind specific user interface controls such as container views to your backend database directly.

Working with Cursors

When results are returned from a SQL query, you often access them using a `Cursor` found in the `android.database.Cursor` class. `Cursor` objects are like file pointers; they allow random access to query results.

You can think of query results as a table, in which each row corresponds to a returned record. The `Cursor` object includes helpful methods for determining how many results were returned by the query the `Cursor` represents and methods for determining the column names (fields) for each returned record. The columns in the query results are defined by the query, not necessarily by the database columns. These might include calculated columns, column aliases, and composite columns.

`Cursor` objects are generally kept around for a time. If you do something simple (such as get a count of records or in cases when you know you retrieved only a single simple record), you can execute your query and quickly extract what you need; don't forget to close the `Cursor` when you're done, as shown here:

```
// SIMPLE QUERY: select * from tbl_books
Cursor c = mDatabase.query("tbl_books",null,null,null,null,null,null);
// Do something quick with the Cursor here...
c.close();
```

Managing Cursors as Part of the Application Lifecycle

When a `Cursor` returns multiple records, or you do something more intensive, you need to consider running this operation on a thread separate from the UI thread. You also need to manage your `Cursor`.

`Cursor` objects must be managed as part of the application lifecycle. When the application pauses or shuts down, the `Cursor` must be deactivated with a call to the `deactivate()` method, and when the application restarts, the `Cursor` should refresh its data using the `requery()` method. When the `Cursor` is no longer needed, a call to `close()` must be made to release its resources.

As the developer, you can handle this by implementing `Cursor` management calls within the various lifecycle callbacks, such as `onPause()`, `onResume()`, and `onDestroy()`.

If you're lazy, like us, and you don't want to bother handling these lifecycle events, you can hand off the responsibility of managing `Cursor` objects to the parent `Activity` by using the `Activity` method called `startManagingCursor()`. The `Activity` handles the rest, deactivating and reactivating the `Cursor` as necessary and destroying the `Cursor` when the `Activity` is destroyed. You can always begin manually managing the `Cursor` object again later by simply calling `stopManagingCursor()`.

Here we perform the same simple query and then hand over `Cursor` management to the parent `Activity`:

```
// SIMPLE QUERY: select * from tbl_books
Cursor c = mDatabase.query("tbl_books",null,null,null,null,null,null);
startManagingCursor(c);
```

Note that, generally, the managed `Cursor` object is a member variable of the class, in terms of scope. You may notice that the `startManagingCursor()` and `stopManagingCursor()` calls are deprecated. In the context of using data on Android, most databases are exposed as content providers. Using a content provider, one can perform queries similar to these, but on more abstract URIs rather than directly on a database using table names. In doing this, you use the higher-level `query()` method of the `ContentResolver` class rather than directly on the database. The proper current method of doing this in a managed way is through the use of the `CursorLoader` class (`android.content.CursorLoader` for API Level 11 and higher, and in the support package for API Level 4 and higher).

Iterating Rows of Query Results and Extracting Specific Data

You can use the `Cursor` to iterate those results, one row at a time using various navigation methods such as `moveToFirst()`, `moveToNext()`, and `isAfterLast()`.

On a specific row, you can use the `Cursor` to extract the data for a given column in the query results. Because SQLite is not strongly typed, you can always pull fields out as Strings using the `getString()` method, but you can also use the type-appropriate extraction utility function to enforce type safety in your application.

For example, the following method takes a valid `Cursor` object, prints the number of returned results, and then prints some column information (name and number of columns). Next, it iterates through the query results, printing each record.

```
public void logCursorInfo(Cursor c) {
    Log.i(DEBUG_TAG, "*** Cursor Begin *** " + " Results:" +
        c.getCount() + " Columns: " + c.getColumnCount());

    // Print column names
    String rowHeaders = "|| ";
    for (int i = 0; i < c.getColumnCount(); i++) {
        rowHeaders = rowHeaders.concat(c.getColumnName(i) + " || ");
    }

    Log.i(DEBUG_TAG, "COLUMNS " + rowHeaders);

    // Print records
    c.moveToFirst();
    while (c.isAfterLast() == false) {

        String rowResults = "|| ";
        for (int i = 0; i < c.getColumnCount(); i++) {
            rowResults = rowResults.concat(c.getString(i) + " || ");
        }

        Log.i(DEBUG_TAG,
            "Row " + c.getPosition() + ": " + rowResults);

       c.moveToNext();
    }
    Log.i(DEBUG_TAG, "*** Cursor End ***");
}
```

The output to the LogCat for this function might look something like Figure 3.1.

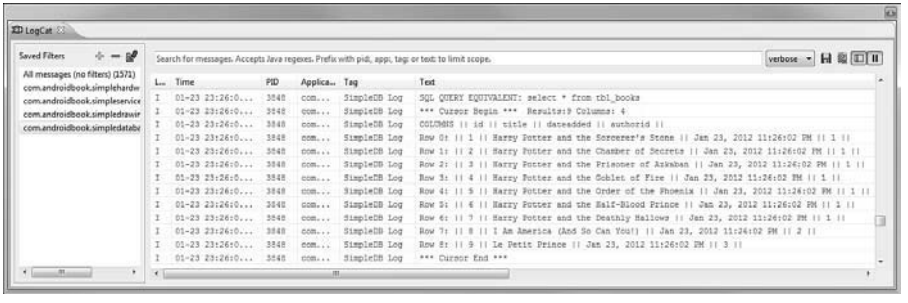

Figure 3.1 Sample log output for the `logCursorInfo()` method.

Executing Simple Queries

Your first stop for database queries should be the `query()` methods available in the `SQLiteDatabase` class. This method queries the database and returns any results as in a `Cursor` object. The `query()` method we mainly use takes the following parameters:

- [`String`]: The name of the table to compile the query against
- [`String Array`]: List of specific column names to return (use `null` for all)
- [`String`] The `WHERE` clause: Use `null` for all; might include selection args as `?`s
- [`String Array`]: Any selection argument values to substitute in for the `?`s in the earlier parameter
- [`String`] `GROUP BY` clause: `null` for no grouping
- [`String`] `HAVING` clause: `null` unless `GROUP BY` clause requires one
- [`String`] `ORDER BY` clause: If `null`, default ordering used
- [`String`] `LIMIT` clause: If `null`, no limit

Previously, we called the `query()` method with only one parameter set to the table name, as shown in the following code:

```
Cursor c = mDatabase.query("tbl_books",null,null,null,null,null,null);
```

This is equivalent to the SQL query

```
SELECT * FROM tbl_books;
```

Tip

The individual parameters for the clauses (`WHERE`, `GROUP BY`, `HAVING`, `ORDER BY`, `LIMIT`) are all `Strings`, but you do not need to include the keyword, such as `WHERE`. Instead, you include the part of the clause after the keyword.

Add a `WHERE` clause to your query, so you can retrieve one record at a time:

```
Cursor c = mDatabase.query("tbl_books", null,
    "id=?", new String[]{"9"}, null, null, null);
```

This is equivalent to the SQL query

```
SELECT * tbl_books WHERE id=9;
```

Selecting all results might be fine for tiny databases, but it is not terribly efficient. You should always tailor your SQL queries to return only the results you require with no extraneous information included. Use the powerful language of SQL to do the heavy lifting for you whenever possible, instead of programmatically processing results yourself. For example, if you need only the titles of each book in the book table, you might use the following call to the `query()` method:

```
String asColumnsToReturn[] = { "title", "id" };
String strSortOrder = "title ASC";
```

```
Cursor c = mDatabase.query("tbl_books", asColumnsToReturn,
    null, null, null, null, strSortOrder);
```

This is equivalent to the SQL query

```
SELECT title, id FROM tbl_books ORDER BY title ASC;
```

Executing More Complex Queries Using `SQLiteQueryBuilder`

As your queries get more complex and involve multiple tables, you should leverage the `SQLiteQueryBuilder` convenience class, which can build complex queries (such as joins) programmatically.

When more than one table is involved, you need to make sure you refer to columns in a table by their fully qualified names. For example, the title column in the `tbl_books` table is `tbl_books.title`. Here we use a `SQLiteQueryBuilder` to build and execute a simple `INNER JOIN` between two tables to get a list of books with their authors:

```
import android.database.sqlite.SQLiteQueryBuilder;
...
SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();

queryBuilder.setTables("tbl_books, tbl_authors");
queryBuilder.appendWhere("tbl_books.authorid=tbl_authors.id");

String asColumnsToReturn[] = {
    "tbl_books.title",
    "tbl_books.id",
    "tbl_authors.firstname",
    "tbl_authors.lastname",
    "tbl_books.authorid" };
String strSortOrder = "title ASC";

Cursor c = queryBuilder.query(mDatabase, asColumnsToReturn,
    null, null, null, null,strSortOrder);
```

First, we instantiate a new `SQLiteQueryBuilder` object. Then we can set the tables involved as part of our `JOIN` and the `WHERE` clause that determines how the `JOIN` occurs. Then, we call the `query()` method of the `SQLiteQueryBuilder` that is similar to the `query()` method we have been using, except we supply the `SQLiteDatabase` instance instead of the table name. The earlier query built by the `SQLiteQueryBuilder` is equivalent to the SQL query:

```
SELECT tbl_books.title,
tbl_books.id,
tbl_authors.firstname,
tbl_authors.lastname,
tbl_books.authorid
FROM tbl_books
INNER JOIN tbl_authors on tbl_books.authorid=tbl_authors.id
ORDER BY title ASC;
```

Executing Raw Queries Without Builders and Column-Mapping

All these helpful Android query utilities can sometimes make building and performing a nonstandard or complex query too verbose. In this case, you might want to consider the `rawQuery()` method. The `rawQuery()` method simply takes a SQL statement `String` (with optional selection arguments if you include `?`s) and returns a `Cursor` of results. If you know your SQL and you don't want to bother learning the ins and outs of all the different SQL query building utilities, this is the method for you.

For example, let's say we have a `UNION` query. These types of queries are feasible with the `QueryBuilder`, but their implementation is cumbersome when you start using column aliases and the like.

Let's say we want to execute the following `SQL UNION` query, which returns a list of all book titles and authors whose names contain the substring `ow` (that is Hallows, Rowling), as in the following:

```
SELECT title AS Name,
'tbl_books' AS OriginalTable
FROM tbl_books
WHERE Name LIKE '%ow%'
UNION
SELECT (firstname||' '|| lastname) AS Name,
'tbl_authors' AS OriginalTable
FROM tbl_authors
WHERE Name LIKE '%ow%'
ORDER BY Name ASC;
```

We can easily execute this by making a string that looks much like the original query and executing the `rawQuery()` method, as shown in the following code:

```
String sqlUnionExample = "SELECT title AS Name, 'tbl_books' AS
    OriginalTable from tbl_books WHERE Name LIKE ? UNION SELECT
    (firstname||' '|| lastname) AS Name, 'tbl_authors' AS OriginalTable
    from tbl_authors WHERE Name LIKE ? ORDER BY Name ASC;";

Cursor c = mDatabase.rawQuery(sqlUnionExample,
    new String[]{ "%ow%", "%ow%"});
```

We make the substrings (`ow`) into selection arguments, so we can use this same code to look for other substrings' searches.

Closing and Deleting a SQLite Database

Although you should always close a database when you are not using it, you might on occasion also want to modify and delete tables and delete your database.

Deleting Tables and Other SQLite Objects

You delete tables and other SQLite objects in exactly the same way you create them. Format the appropriate SQLite statements and execute them. For example, to drop our tables and triggers, we can execute three SQL statements:

```
mDatabase.execSQL("DROP TABLE tbl_books;");
mDatabase.execSQL("DROP TABLE tbl_authors;");
mDatabase.execSQL("DROP TRIGGER IF EXISTS fk_insert_book;");
```

Closing a SQLite Database

You should close your database when you are not using it. You can close the database using the `close()` method of your `SQLiteDatabase` instance, like this:

```
mDatabase.close();
```

Deleting a SQLite Database Instance Using the Application Context

The simplest way to delete a `SQLiteDatabase` is to use the `deleteDatabase()` method of your application `Context`. You delete databases by name and the deletion is permanent. You lose all data and schema information.

```
deleteDatabase("my_sqlite_database.db");
```

Designing Persistent Databases

Generally speaking, an application creates a database and uses it for the rest of the application's lifetime—by which we mean until the application is uninstalled from the device. So far, we've talked about the basics of creating a database, using it, and then deleting it.

In reality, most mobile applications do not create a database on-the-fly, use them, and then delete them. Instead, they create a database the first time they need it and then use it. The Android SDK provides a helper class called `SQLiteOpenHelper` to help you manage your application's database.

To create a SQLite database for your Android application using the `SQLiteOpenHelper`, you need to extend that class and then instantiate an instance of it as a member variable for use in your application. To illustrate how to do this, let's create a new Android project called PetTracker.

Tip

Many of the code examples provided in this section are taken from the PetTracker application. The source code for the PetTracker application is provided for download on the book's websites.

Keeping Track of Database Field Names

You've probably realized by now that it is time to start organizing your database fields programmatically to avoid typos and such in your SQL queries. One easy way you do this is to make a class to encapsulate your database schema in a class, such as `PetDatabase`, shown here:

```
import android.provider.BaseColumns;


public final class PetDatabase {

    private PetDatabase() {}

    public static final class Pets implements BaseColumns {
        private Pets() {}
        public static final String PETS_TABLE_NAME="table_pets";
        public static final String PET_NAME="pet_name";
        public static final String PET_TYPE_ID="pet_type_id";
        public static final String DEFAULT_SORT_ORDER="pet_name ASC";
    }

    public static final class PetType implements BaseColumns {
        private PetType() {}
        public static final String PETTYPE_TABLE_NAME="table_pettypes";
        public static final String PET_TYPE_NAME="pet_type";
        public static final String DEFAULT_SORT_ORDER="pet_type ASC";
    }
}
```

By implementing the `BaseColumns` interface, we begin to set up the underpinnings for using database-friendly user interface controls in the future, which often require a specially named column called `_id` to function properly. We rely on this column as our primary key.

Extending the `SQLiteOpenHelper` Class

To extend the `SQLiteOpenHelper` class, we must implement several important methods, which help manage the database versioning. The methods to override are `onCreate()` and `onUpgrade()` and optionally `onDowngrade()` and `onOpen()`. We use our newly defined `PetDatabase` class to generate appropriate SQL statements, as shown here:

```
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

import com.androidbook.PetTracker.PetDatabase.PetType;
import com.androidbook.PetTracker.PetDatabase.Pets;
```

```
class PetTrackerDatabaseHelper extends SQLiteOpenHelper {

    private static final String DATABASE_NAME = "pet_tracker.db";
    private static final int DATABASE_VERSION = 1;

    PetTrackerDatabaseHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL("CREATE TABLE " +PetType.PETTYPE_TABLE_NAME+" ("
            + PetType._ID + " INTEGER PRIMARY KEY AUTOINCREMENT ,"
            + PetType.PET_TYPE_NAME + " TEXT"
            + ");");
        db.execSQL("CREATE TABLE " + Pets.PETS_TABLE_NAME + " ("
            + Pets._ID + " INTEGER PRIMARY KEY AUTOINCREMENT ,"
            + Pets.PET_NAME + " TEXT,"
            + Pets.PET_TYPE_ID + " INTEGER" // FK to pet type table
            + ");");
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion,
        int newVersion){
        // Housekeeping here.
        // Implement how to "move" your application data
        // during an upgrade of schema versions.
        // Move or delete data as required. Your call.
    }

    @Override
    public void onOpen(SQLiteDatabase db) {
        super.onOpen(db);
    }
}
```

Now we can create a member variable for our database like this:

```
PetTrackerDatabaseHelper mDatabase = new
    PetTrackerDatabaseHelper(this.getApplicationContext());
```

Now, whenever our application needs to interact with its database, we request a valid database object. We can request a read-only database or a database that we can also write to. We can also close the database. For example, here we get a database we can write data to:

```
SQLiteDatabase db = mDatabase.getWritableDatabase();
```

Binding Data to the Application User Interface

In many cases with application databases, you want to couple your user interface with the data in your database. You might want to fill drop-down lists with values from a database table, or fill out form values, or display only certain results. There are various ways to bind database data to your user interface. You, as the developer, can decide whether to use built-in data-binding functionality provided with certain user interface controls, or build your own user interfaces from the ground up.

Working with Database Data Like Any Other Data

If you peruse the PetTracker application provided on the book's websites, you notice that its functionality includes no magical data-binding features, yet the application clearly uses the database as part of the user interface.

Specifically, the database is leveraged:

- When you fill out the Pet Type field, the `AutoComplete` feature is seeded with pet types already in listed in the `table_pettypes` table (Figure 3.2, left).
- When you save new records using the Pet Entry Form (Figure 3.2, middle).
- When you display the Pet List screen, you query for all pets and use a `Cursor` to programmatically build a `TableLayout` on-the-fly (Figure 3.2, right).

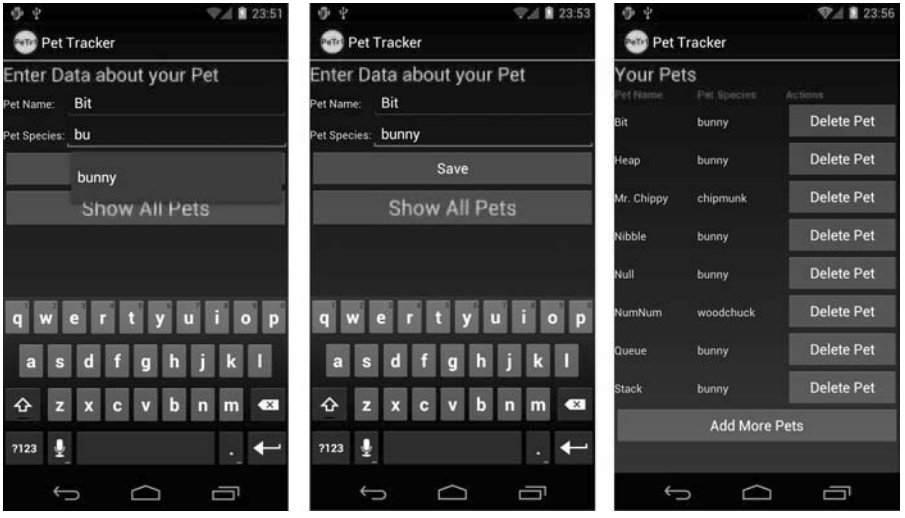

Figure 3.2 The PetTracker application: Entry Screen (left, middle) and Pet Listing Screen (right).

This might work for small amounts of data; however, there are various drawbacks to this method. For example, all the work is done on the main thread, so the more records

you add, the slower your application response time becomes. Second, there's quite a bit of custom code involved to map the database results to the individual user interface components. If you decide you want to use a different control to display your data, you have quite a lot of rework to do. Third, we constantly requery the database for fresh results, and we might be requerying far more than necessary.

Note

Yes, we really named our pet bunnies after data structures and computer terminology. We are that geeky. Null, for example, is a rambunctious little black bunny. Shane enjoys pointing at him and calling himself a Null pointer.

Binding Data to Controls Using Data Adapters

Ideally, you'd like to bind your data to user interface controls and let them take care of the data display. For example, we can use a fancy `ListView` to display the pets instead of building a `TableLayout` from scratch. We can spin through our `Cursor` and generate `ListView` child items manually, or even better, we can simply create a data adapter to map the `Cursor` results to each `TextView` child within the `ListView`.

The PetTracker2 application behaves much like the PetTracker sample application, except that it uses the `SimpleCursorAdapter` with `ListView` and an `ArrayAdapter` to handle `AutoCompleteTextView` features.

Tip

The source code for subsequent upgrades to the series of PetTracker applications is provided for download on the book's websites.

Binding Data Using `SimpleCursorAdapter`

Let's now look at how we can create a data adapter to mimic our Pet Listing screen, with each pet's name and species listed. We also want to continue to have the ability to delete records from the list.

A `ListView` container can contain children such as `TextView` objects. In this case, we want to display each Pet's name and type. We therefore create a layout file called `pet_item.xml` that becomes our `ListView` item template:

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

```
        android:layout_height="?android:attr/listPreferredItemHeight"
        android:layout_alignParentLeft="true" />
    <TextView
        android:id="@+id/TextView_PetType"
        android:layout_width="wrap_content"
        android:layout_height="?android:attr/listPreferredItemHeight"
        android:layout_alignParentRight="true" />
</RelativeLayout>
```

Next, in our main layout file for the Pet List, we place our `ListView` in the appropriate place on the overall screen. The `ListView` portion of the layout file might look something like this:

```
<ListView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/petList" android:divider="#000" />
```

Now to programmatically fill our `ListView`, we must take the following steps:

1. Perform our query and return a valid `Cursor` (a member variable).
2. Create a data adapter that maps the `Cursor` columns to the appropriate `TextView` controls within our `pet_item.xml` layout template.
3. Attach the adapter to the `ListView`.

In the following code, we perform these steps:

```
SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();
queryBuilder.setTables(Pets.PETS_TABLE_NAME +", " +
    PetType.PETTYPE_TABLE_NAME);

queryBuilder.appendWhere(Pets.PETS_TABLE_NAME + "." +
    Pets.PET_TYPE_ID + "=" + PetType.PETTYPE_TABLE_NAME + "." +
    PetType._ID);

String asColumnsToReturn[] = { Pets.PETS_TABLE_NAME + "." +
    Pets.PET_NAME, Pets.PETS_TABLE_NAME +
    "." + Pets._ID, PetType.PETTYPE_TABLE_NAME + "." +
    PetType.PET_TYPE_NAME };

mCursor = queryBuilder.query(mDB, asColumnsToReturn, null, null,
    null, null, Pets.DEFAULT_SORT_ORDER);

startManagingCursor(mCursor);

ListAdapter adapter = new SimpleCursorAdapter(this,
    R.layout.pet_item, mCursor,
```

```
    new String[]{Pets.PET_NAME, PetType.PET_TYPE_NAME},
    new int[]{R.id.TextView_PetName, R.id.TextView_PetType });

ListView av = (ListView)findViewById(R.id.petList);
av.setAdapter(adapter);
```

Notice that the `_id` column and the expected name and type columns appear in the query. This is required for the adapter and `ListView` to work properly.

Using a `ListView` (Figure 3.3, left) instead of a custom user interface enables us to take advantage of the `ListView` control's built-in features, such as scrolling when the list becomes longer, and the ability to provide context menus as needed. The `_id` column is used as the unique identifier for each `ListView` child node. If we choose a specific item on the list, we can act on it using this identifier, for example, to delete the item.

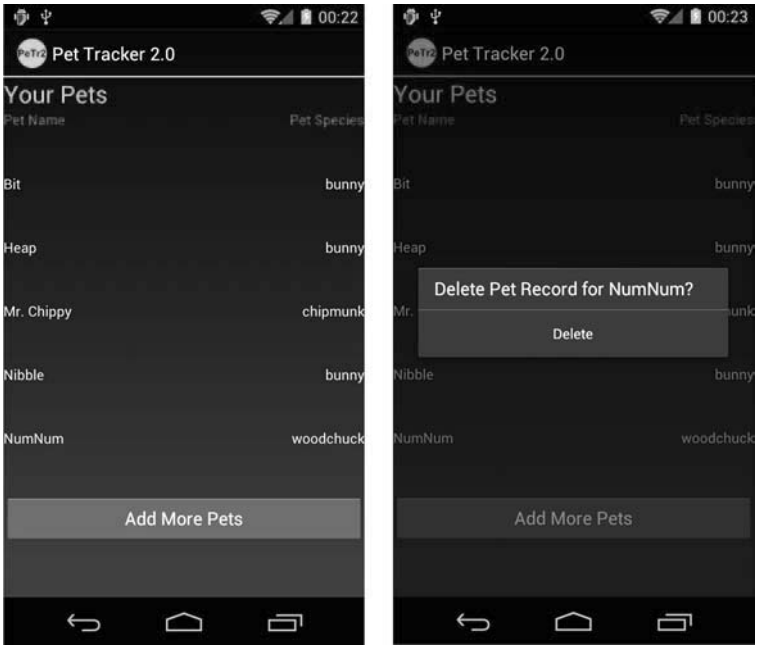

Figure 3.3 The PetTracker2 application: Pet Listing Screen `ListView` (left) with Delete feature (right).

Now we reimplement the Delete functionality by listening for `onItemClick()` events and providing a Delete Confirmation dialog (Figure 3.3, right):

```
av.setOnItemClickListener(new AdapterView.OnItemClickListener() {
    public void onItemClick(AdapterView<?> parent, View view,
            int position, long id) {
        final long deletePetId = id;
```

```
        RelativeLayout item = (RelativeLayout) view;
        TextView nameView = (TextView) item
                .findViewById(R.id.TextView_PetName);
        String name = nameView.getText().toString();
        new AlertDialog.Builder(PetTrackerListActivity.this)
                .setMessage("Delete Pet Record for " + name + "?")
                .setPositiveButton("Delete",
                        new DialogInterface.OnClickListener() {
                            public void onClick(DialogInterface dialog,
                                    int which) {

                                deletePet(deletePetId);
                                mCursor.requery();
                            }
                        }).show();
    }
});
```

Note that within the PetTracker2 sample application, we also use an `ArrayAdapter` to bind the data in the `pet_types` table to the `AutoCompleteTextView` on the Pet Entry screen. Although our next example shows you how to do this in a preferred manner, we left this code in the PetTracker sample to show you that you can always intercept the data your `Cursor` provides and do what you want with it. In this case, we create a `String` array for the AutoText options by hand. We use a built-in Android layout resource called `android.R.layout.simple_dropdown_item_1line` to specify what each individual item within the AutoText listing looks like. You can find the built-in layout resources provided within your appropriate Android SDK version's resource subdirectory.

A Note on Design

In this example, we've followed the traditional Android use of context menus with the press-and-hold, as the SDK provides for. However, in Android 4.0, the design guidelines have changed to recommend using press-and-hold for selection. See http://developer.android.com/design/patterns/new-4-0.html (http://goo.gl/aBH6n) for more information. When you are using contextual menus, be sure the dialog contains the context of what item the action is being taken on, such as the name as we've shown here.

Storing Nonprimitive Types (Such as Images) in the Database

Because SQLite is a single file, it makes little sense to try to store binary data in the database. Instead store the *location* of data, as a file path or a URI in the database, and access it appropriately.

Summary

There are a variety of different ways to store and manage application data on the Android platform. The method you use depends on what kind of data you need to store. Application-specific SQLite databases are secure and efficient mechanisms for structured data storage. You now know how to design persistent data-access mechanisms in your Android application, and you also learned how to bind data from various sources to user interface controls, such as `ListView` objects.

References and More Information

Android Dev Guide: "Data Storage":
http://d.android.com/guide/topics/data/data-storage.html
Android SDK Documentation for the `android.database.sqlite` package:
http://d.android.com/reference/android/database/sqlite/package-summary.html
SQLite website:
http://www.sqlite.org/index.html
SQLzoo.net:
http://sqlzoo.net/

This page intentionally left blank

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X-Y

Z