This is from the book: **Android Wireless Application Development:**

(to access the book login sslvpn.usc.edu**, in the URL there enter this** http://my.safaribooksonline.com/9780321619686/firstchapter**)**

Appendix D:

D. The SQLite Quick-Start Guide

The Android System allows individual applications to have private SQLite databases in which to store their application data. This Quick-Start Guide is not a complete documentation of the SQLite commands. Instead, it is designed to get you up and running with common tasks. The first part of this appendix introduces the features of the `sqlite3` command-line tool. We then provide an in-depth database example using many common SQLite commands. See the online SQLite documentation (www.sqlite.org) for a complete list of features, functionality, and limitations of SQLite.

Exploring Common Tasks with SQLite

SQLite is a lightweight and compact, yet powerful, embedded relational database engine available as public domain. It is fast and has a small footprint, making it perfect for phone system use. Instead of the heavyweight server-based databases such as Oracle and Microsoft SQL Server, each SQLite database is contained within a self-contained single file on disk.

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

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

For example, the database for the `PetTracker` application provided in this book is found at

`/data/data/com.androidbook.PetTracker/databases/pet_tracker.db`

The database file format is standard and can be moved across platforms. You can use the Dalvik Debug Monitor Service (DDMS) File Explorer to pull the database file and inspect it with third-party tools, if you like.

Application-specific SQLite databases are private and accessible only from within that application. To expose application data to other applications, the application must become a content provider. Content providers are covered in Chapter 9 ***(attached in the end of the document)***, "Using Android Data and Storage APIs."

Using the `sqlite3` Command-Line Interface

In addition to programmatic access to create and use SQLite databases from within your applications, which we discuss in Chapter 9, you can also interact with the database using the familiar command-line `sqlite3` tool, which is accessible via the Android Debug Bridge (ADB) remote shell.

The command-line interface for SQLite, called `sqlite3`, is exposed using the ADB tool, which we covered in Appendix C, "The Android Debug Bridge Quick-Start Guide."

Launching the `sqlite3` Command-Line Interface and Connecting to a Database

You must launch ADB shell interface on the specific emulator or device to use the `sqlite3` commands.

Launching the ADB Shell

If only one Android device (or emulator) is running, you can connect by simply typing

```
c:\>adb shell
```

If you want to connect to a specific instance of the emulator, you can connect by typing

```
adb –s <serialNumber> shell
```

For example, to connect to the emulator at port 5554, you would use the following command:

```
adb –s emulator-5554 shell
```

For more information on how to determine the serial number of an emulator or device instance, please see Appendix C.

Connecting to the Database

Now you can connect to the Android application database of your choice by name. For example, to connect to the database we created with the Pet Tracker application, we would connect like this:

```
c:\>adb shell
# sqlite3 /data/data/com.androidbook.PetTracker/databases/pet_tracker.db
SQLite version 3.5.9
Enter ".help" for instructions
sqlite>
```

Now we have the `sqlite3` command prompt, where we can issue commands. You can exit the interface at any time by typing

```
sqlite>.quit
```

or

```
sqlite>.exit
```

Exploring Your Database

You can use the `sqlite3` commands to explore what your database looks like and interact with it. You can

- List available databases
- List available tables
- View all the indices on a given table
- Show the database schema

Listing Available Databases

You can list the names and file locations attached to this database instance. Generally, you have your main database and a temp database, which contains temp tables. You can list this information by typing

```
sqlite> .databases
seq  name file
---  ---- -----------------------------------------------
0    main /data/data/com.androidbook.PetTracker/databases/...
1    temp
sqlite>
```

Listing Available Tables

You can list the tables in the database you connect to by typing

```
sqlite> .tables
android_metadata  table_pets        table_pettypes
```

```
sqlite>
```

Listing Indices of a Table

You can list the indices of a given table by typing

```
sqlite>.indices table_pets
```

Listing the Database Schema of a Table

You can list the schema of a given table by typing

```
sqlite>.schema table_pets
CREATE TABLE table_pets (_id INTEGER PRIMARY KEY
AUTOINCREMENT,pet_name TEXT,pet_type_id INTEGER);
sqlite>
```

Listing the Database Schema of a Database

You can list the schemas for the entire database by typing

```
sqlite>.schema
CREATE TABLE android_metadata (locale TEXT);
CREATE TABLE table_pets (_id INTEGER PRIMARY KEY
AUTOINCREMENT,pet_name TEXT,pet_type_id INTEGER);
CREATE TABLE table_pettypes (_id INTEGER PRIMARY KEY
AUTOINCREMENT,pet_type TEXT);
sqlite>
```

Importing and Exporting the Database and Its Data

You can use the `sqlite3` commands to import and export database data and the schema and interact with it. You can

- Send command output to a file instead of to STDOUT (the screen).
- Dump the database contents as a SQL script (so you can re-create it later).
- Execute SQL scripts from files.
- Import data into the database from a file.

Note

The file paths are on the Android device, not your computer. You need to find a directory on the Android device in which you have permission to read-and-write files. For example, `/data/local/tmp/` is a shared directory.

Sending Output to a File

Often, you want the `sqlite3` command results to pipe to a file instead of to the screen. To do this, you can just type the `output` command followed by the file path to write to on the Android system. For example:

```
sqlite>.output /data/local/tmp/dump.sql
```

Dumping Database Contents

You can create a SQL script to create tables and their values by using the `dump` command. The `dump` command creates a transaction, which includes calls to CREATE TABLE and INSERT to populate the database with data. This command can take an optional table name or dump the whole database.

Tip

The `dump` command is a great way to do a full archival backup of your database.

For example, the following commands pipe the dump output for the `table_pets` table to a file and then sets the output mode back to the console.

```
sqlite>.output /data/local/tmp/dump.sql
sqlite>.dump table_pets
sqlite>.output stdout
```

You can then use DDMS and the File Explorer to pull the SQL file off the Android file system. The resulting `dump.sql` file looks like this:

```
BEGIN TRANSACTION;
CREATE TABLE table_pets (
_id INTEGER PRIMARY KEY AUTOINCREMENT,
pet_name TEXT,
pet_type_id INTEGER);

INSERT INTO "table_pets" VALUES(1,'Rover',9);
INSERT INTO "table_pets" VALUES(2,'Garfield',8);
COMMIT;
```

Executing SQL Scripts From Files

You can create SQL script files and run them through the console. These scripts must be on the Android file system. For example, let's put a SQL script called `myselect.sql` in the `/data/local/tmp/` directory of the Android file system.

The file has two lines

```
SELECT * FROM table_pettypes;
SELECT * FROM table_pets;
```

We can then run this SQL script by typing

```
sqlite>.read /data/local/tmp/myselect.sql
```

You see the query results on the command line.

Importing Data

You can import formatted data using the import and separator commands. Files like CSV use commas for delimiters, but other data formats might be spaces or tabs. You specify the delimiter using the separator command. You specify the file to import using the import command.

For example, put a CSV script called `some_data.csv` in the **`/data/local/tmp/`** directory of the Android file system.

The file has four lines. It is a comma-delimited file of pet type ids and pet type names:

```
18,frog
19,turkey
20,piglet
21,great white shark
```

You can then import this data into the `table_pettypes` table, which has two columns: an `_id` column and a pet type name. To import this data, type the following command:

```
sqlite>.separator ,
sqlite>.import /data/local/tmp/some_data.csv table_pettypes
```

Now, if you query the table, you see it has four new rows.

Executing SQL Commands on the Command Line

You can also execute raw SQL commands on the command line. Simply type the SQL command, making sure it ends with a semicolon (;).

If you use queries, you might want to change the output mode to column so that query results are easier to read (in columns) and the headers (column names) are printed.

For example:

```
sqlite> .mode column
sqlite> .header on
sqlite> select * from table_pettypes WHERE _id < 11;
_id         pet_type
----------  ----------
8           bunny
9           fish
10          dog
sqlite>
```

You're not limited to queries, either. You can execute any SQL command you see in a SQL script on the command line if you like.

Tip

We've found it helpful to use the `sqlite3` command line to test SQL queries if `our` Android SQL queries with `QueryBuilder` are not behaving. This is especially true of more complicated queries.

You can also control the width of each column (so text fields don't truncate) using the `width` command. For example, the following command prints query results with the first column 5 characters wide (often an ID column), followed by a second column 50 characters wide (text column).

```
sqlite> .width 5 50
```

Poking Around Within SQLite Internals

SQLite keeps the database schema in a special table called `sqlite_master`. You should consider this table read-only. SQLite stores temporary tables in a special table called `sqlite_temp_master`, which is also a temporary table.

Using Other `sqlite3` Commands

A complete list of `sqlite3` commands is available by typing

```
sqlite> .help
```

Understanding SQLite Limitations

SQLite is powerful, but it has several important limitations compared to traditional SQL Server implementations, such as the following:

- SQLite is not a substitute for a high-powered, server-driven database.
- Being file-based, the database is meant to be accessed in a serial, not a concurrent manner. Think "single user"—the Android application. It has some concurrency features, but they are limited.
- Access control is maintained by file permissions, not database user permissions.
- Referential integrity is not maintained. For example, FOREIGN KEY constraints are parsed (for example, in CREATE TABLE) but not enforced automatically. However, using TRIGGER functions can enforce them.
- ALTER TABLE support is limited. You can use only RENAME TABLE and ADD COLUMN. You may not drop or alter columns or perform any other such operations. This can make database upgrades a bit tricky.
- TRIGGER support is limited. You cannot use: `FOR EACH STATEMENT` or `INSTEAD OF`. You cannot create recursive triggers.
- You cannot nest TRANSACTION operations.
- VIEWs are read-only.
- No RIGHT OUTER JOINs or FULL OUTER JOINs.
- SQLite does not support STORED PROCEDUREs or auditing.
- The built-in FUNCTIONs of the SQL language are limited.
- For limitations on the maximum database size, table size, and row size, see the SQLite documentation, including the helpful Omitted SQL pagewww.sqlite.org/omitted.html and the Unsupported SQL Wiki page www.sqlite.org/cvstrac/wiki?p=UnsupportedSql.

Understanding SQLite By Example: Student Grade Database

Let's work through a Teacher "Grades" example database to show standard SQL commands to create and work with a database. Although you can create this database using the `sqlite3` command line, we suggest using the Android application to create the empty Grades database, so that it is created in a standard "Android" way.

The setup: The purpose of the Student grade database is to keep track of each student's test results for the class. Each student's grade is calculated from the performance on four quizzes, one Midterm and one Final. All tests are graded on a scale of 0–100.

Designing the Student Grade Database Schema

The Teacher "Grades" database has three tables: Students, Tests, and TestResults.

The Student table contains student information. The Tests table contains information about each test and how much it counts toward the student's overall grade. Finally, all students' test results are stored in the TestResults table.

Setting Column Datatypes

`SQLite3` has support for the following common datatypes for columns:

- INTEGER (signed integers)
- REAL (floating point values)

- TEXT (UTF-8 or UTF-16 string; encoded using database encoding)
- BLOB (data chunk)

Tip

Do not store files like images in the database. Instead, store them as files in the application file directory and store the filename or URI path in the database.

Creating Simple Tables with AUTOINCREMENT

First, let's create the Students table. We want a student id to reference each student. We can make this the primary key and have it autoincrement. We also want the first and last name of each student, and we require these fields (no nulls). Here's our SQL statement:

```
CREATE TABLE Students (
id INTEGER PRIMARY KEY AUTOINCREMENT,
fname TEXT NOT NULL,
lname TEXT NOT NULL );
```

For the Tests table, we want a test id to reference each test or quiz, much like the Students table. We also want a friendly name for each test and a weight value for how much each test counts for the student's final grade (as a percentage). Here's our SQL statement:

```
CREATE TABLE Tests (
id INTEGER PRIMARY KEY AUTOINCREMENT,
testname TEXT,
weight REAL DEFAULT .10 CHECK (weight<=1));
```

Inserting Data into Tables

Before we move on, let's add some data to these tables. To add a record to the Students table, you need to specify the column names and the values in order. For example:

```
INSERT into Students
(fname, lname)
VALUES
('Harry', 'Potter');
```

Now, we're going to add a few more records to this table for Ron and Hermione. At the same time, we need to add a bunch of records to the Tests table. First we add the Midterm, which counts for 25 percent of the grade (and also a Final for 35 percent):

```
INSERT into Tests
(testname, weight)
VALUES
('Midterm', .25);
```

Then we add a couple quizzes, which use the default weight of 10 percent:

```
INSERT into Tests (testname) VALUES ('Quiz 1');
```

Finally, we add a final worth 35 percent of the total grade.

Querying Tables for Results with SELECT

How do we know the data we've added is in the table? Well, that's easy. We simply query for all rows in a table using a SELECT:

```
SELECT * FROM Tests;
```

This returns all records in the Tests table:

```
id     testname         weight
-----  ---------------  ------
1      Midterm          0.25
2      Quiz 1           0.1
3      Quiz 2           0.1
4      Quiz 3           0.1
5      Quiz 4           0.1
6      Final            0.35
```

Now, ideally, we want the weights to add up to 1.0. Let's check using the SUM aggregate function to sum all the weight values in the table:

```
SELECT SUM(weight) FROM Tests;
```

This returns the sum of all weight values in the Tests table:

```
SUM(weight)
-----------
1.0
```

We can also create our own columns and alias them. For example, we can create a column alias called `fullname` that is a calculated column: It's the student's first and last names concatenated using the `||` concatenation.

```
SELECT fname||' '|| lname AS fullname, id FROM Students;
```

This gives us the following results:

```
fullname          id
------------      --
Harry Potter       1
Ron Weasley        2
Hermione Granger   3
```

Creating Tables with Foreign Keys and Composite Primary Keys

Now that we have our students and tests all set up, let's create the TestResults table. This is a more complicated table. It's a list of student-test pairings, along with the score.

The TestResults table pairs up Student ids from the Student table with Test ids from the Tests table. Columns, which link to other tables in this way, are often called foreign keys. We want unique student-test pairings, so then we create a composite primary key from the student and test foreign keys. Lastly, we enforce that the scores are whole numbers between 0 and 100. No extra credit or retaking tests in this class!

```
CREATE TABLE TestResults (
studentid INTEGER REFERENCES Students(id),
testid INTEGER REFERENCES Tests(id),
score INTEGER CHECK (score<=100 AND score>=0),
PRIMARY KEY (studentid, testid));
```

Tip

SQLite does not enforce foreign key constraints, but you can set them up anyway and enforce the constraints by creating triggers. For an example of using triggers to enforce foreign key constraints in SQL, check out the FullDatabase project provided on the CD for Chapter 9.

Now it's time to insert some data into this table. Let's say Harry Potter received an 82 percent on the Midterm:

```
INSERT into TestResults
(studentid, testid, score)
VALUES
(1,1,82);
```

Now let's input the rest of the student's scores. Harry is a good student. Ron is not a good student, and Hermione aces every test (of course). When they're all added, we can list them. We can do a SELECT * to get all columns, or we can specify the columns we want explicitly like this:

```
SELECT studentid, testid, score FROM TestResults;
```

Here are the results from this query:

```
studentid   testid      score
----------  ----------  -----
1           1           82
1           2           88
1           3           78
1           4           90
1           5           85
1           6           94
2           1           10
2           2           90
2           3           50
2           4           55
2           5           45
2           6           65
3           6           100
3           5           100
3           4           100
3           3           100
3           2           100
3           1           100
```

Altering and Updating Data in Tables

Ron's not a good student, and yet he received a 90 percent on Quiz #1. This is suspicious, so as the teacher, we check the actual paper test to see if we made a recording mistake. He actually earned 60 percent. Now we need to update the table to reflect the correct score:

```
UPDATE TestResults
SET score=60
WHERE studentid=2 AND testid=2;
```

You can delete rows from a table using the DELETE function. For example, to delete the record we just updated:

```
DELETE FROM TestResults WHERE studentid=2 AND testid=2;
```

You can delete all rows in a table by not specifying the WHERE clause:

```
DELETE FROM TestResults;
```

Querying for Information Stored in Multiple Tables Using JOIN

Now that we have all our data in our database, it is time to use it. The preceding listing was not easy for a human to read. It would be much nicer to see a listing with the names of the Students and Names of the tests instead of their ids.

Combining data is often handled by performing a JOIN with multiple table sources; there are different kinds of JOINS. When you work with multiple tables, you need to specify which table a column belongs to (especially with all these different id columns). You can refer to columns by their column name or by their table name, then a dot (.), and then the column name.

Let's relist the grades again, only this time, include the name of the test and the name of the student. Also, we limit our results only to the score for the Final (`test id 6`):

```
SELECT
Students.fname||' '|| Students.lname AS StudentName,
Tests.testname,
TestResults.score
FROM TestResults
JOIN Students
        ON (TestResults.studentid=Students.id)
JOIN Tests
        ON (TestResults.testid=Tests.id)
WHERE testid=6;
```

which gives us the following results (you could leave off the WHERE to get all tests):

```
StudentName         testname        score
------------------  --------------  -----
Harry Potter        Final           94
Ron Weasley         Final           65
Hermione Granger    Final           100
```

Using Calculated Columns

Hermoine always likes to know where she stands. When she comes to ask what her final grade is likely to be, we can perform a single query to show all her results and calculate the weighted scores of all her results:

```
SELECT
Students.fname||' '|| Students.lname AS StudentName,
Tests.testname,
Tests.weight,
TestResults.score,
(Tests.weight*TestResults.score) AS WeightedScore
FROM TestResults
JOIN Students
        ON (TestResults.studentid=Students.id)
JOIN Tests
        ON (TestResults.testid=Tests.id)
WHERE studentid=3;
```

This gives us predictable results:

```
StudentName         testname  weight  score  WeightedScore
----------------    --------  ------  -----  ------------
Hermione Granger    Midterm   0.25    100         25.0
Hermione Granger    Quiz 1    0.1     100         10.0
Hermione Granger    Quiz 2    0.1     100         10.0
Hermione Granger    Quiz 3    0.1     100         10.0
Hermione Granger    Quiz 4    0.1     100         10.0
Hermione Granger    Final     0.35    100         35.0
```

We can just add up the Weighed Scores and be done, but we can also do it via the query:

```
SELECT
Students.fname||' '|| Students.lname AS StudentName,
```

```
SUM((Tests.weight*TestResults.score)) AS TotalWeightedScore
FROM TestResults
JOIN Students
        ON (TestResults.studentid=Students.id)
JOIN Tests
        ON (TestResults.testid=Tests.id)
WHERE studentid=3;
```

Here we get a nice consolidated listing:

```
StudentName       TotalWeightedScore
---------------   -----------------
Hermione Granger  100.0
```

If we wanted to get all our students' grades, we need to use the GROUP BY clause. Also, let's order them so the best students are at the top of the list:

```
SELECT
Students.fname||' '|| Students.lname AS StudentName,
SUM((Tests.weight*TestResults.score)) AS TotalWeightedScore
FROM TestResults
JOIN Students
        ON (TestResults.studentid=Students.id)
JOIN Tests
        ON (TestResults.testid=Tests.id)
GROUP BY TestResults.studentid
ORDER BY TotalWeightedScore DESC;
```

This makes our job as teacher almost too easy, but at least we're saving trees by using a digital grade book.

```
StudentName               TotalWeightedScore
------------------------  -----------------
Hermione Granger          100.0
Harry Potter              87.5
Ron Weasley               46.25
```

Using Subqueries for Calculated Columns

You can also include queries within other queries. For example, you can list each Student and a count of how many tests they "passed," in which passing is getting a score higher than 60, as in the following:

```
SELECT
Students.fname||' '|| Students.lname AS StudentName,
Students.id AS StudentID,
(SELECT COUNT(*)
FROM TestResults
WHERE TestResults.studentid=Students.id
AND TestResults.score>60)
AS TestsPassed
FROM Students;
```

Again, we see that Ron needs a tutor:

```
StudentName        StudentID  TestsPassed
-----------        ---------  -----------
Harry Potter       1          6
Ron Weasley        2          1
Hermione Granger   3          6
```

Deleting Tables

You can always delete tables using the DROP TABLE command. For example, to delete the TestResults table, use the following SQL command:

```
DROP TABLE TestResults;
```

CHAPTER 9: FROM THE BOOK

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.

These databases and the data within 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 (discussed later in this chapter).

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. There are also specialized `Cursor` objects for iterating query results. You can also find all the specialized exceptions associated with SQLite.

Here we focus on creating databases within 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 (within 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 accessible through the ADB shell interface. This can be an extremely helpful debugging tool for developers and quality assurance personnel, who might want to manage the database state (and content) for testing purposes.

Tip

For more information about designing SQLite databases and interacting with them via the command line tool, please see Appendix D, "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 SQL (or SQLite) or a bit rusty on your syntax, this tutorial is for you.

Creating a SQLite Database

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

Tip

You can find the FullDatabase project on the CD provided at the end of this book or online at the book Web site. This project contains a single class that creates a SQLite database, populates it with several tables worth of data, queries and manipulates the data in various ways, and then deletes the database.

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's Database File on the Device File System

Android applications store their databases (SQLite or otherwise) under 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.FullDatabase/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);
```

This `CREATE TABLE` SQL statement can be encapsulated in a static final String variable (called `CREATE_AUTHOR_TABLE`) and then executed on our 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:

Code View: Scroll / Show All

```
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:

Code View: Scroll / Show All

```
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 `id` of the newly created record. We use this author `id` 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 are not working and you'd really like to know why.

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. Notice that we did this in the FullDatabase sample project.

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 will be substituted in place of the ?s from the second parameter.

Passing `null` to the WHERE clause modifies all records within the table. This 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` corresponding 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 will be substituted in place of the ?s from the second parameter.

Passing `null` to the WHERE clause deletes all records within the table. For example, this function call deletes all records within 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 identifier. The following function takes a parameter `bookId` and deletes the record corresponding to that unique `id` (primary key) within 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 `id`.

```
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 rollback 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, updated 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 will be rolled back and not committed. Finally, you end your transaction by calling `endTransaction()`. 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, while 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 widgets such as container views to your backend database directly.

Working with Cursors

When results are returned from a SQL query, they are often accessed using a `Cursor` found in the `android.database.Cursor` class. `Cursor` objects are rather 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 and 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 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` will handle 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` is a member variable of the class, scope-wise.

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.

Code View: Scroll / Show All

```
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 9.1.

Figure 9.1. Sample log output for the `logCursorInfo()` method.
[View full size image]

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 above
- [`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 in the chapter, we called the `query()` method with only one parameter set the table name.

```
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.

So let's add a WHERE clause to our query, so we 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 like Joins Using `SQLiteQueryBuilder`

As your queries get more complex and involve multiple tables, you want to leverage the `SQLiteQueryBuilder` convenience class, which can build complex queries programmatically.

When more than one table is involved, you need to make sure you refer to columns within a table by their fully qualified names. For example, the title column within the `tbl_books` table would be `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 above 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 Bothering with 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 name contains 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.

```
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

Deleting tables and other SQLite objects is done in exactly the same was as creating 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`. Databases are deleted by name and the deletion is permanent. All data and schema information is lost.

```
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 phone. So far, we've talked about the basics of creating a database, using it, and then deleting it. Actually, that is exactly what the "FullDatabase" sample application provided on the CD and book Web site does.

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 within your application. To illustrate how to do this, let's create a new Android project called PetTracker.

Tip

You can find the PetTracker project on the CD provided at the end of this book or online at the book Web site. This project creates and maintains a simple database of pet names and species. The user interface contains two screens: a form to input data and a screen to display the data. No complicated data-binding is performed in this version of the application. We build upon this example in future sections of this chapter.

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 to 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 widgets 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()`, `onUpgrade()`, and `onOpen()`. We use our newly defined `PetDatabase` class to generate appropriate SQL statements, as shown here:

Code View: Scroll / Show All

```
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 "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 widgets, or you can 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 CD, 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 save new records using the Pet Entry Form (Figure 9.2, left).

Figure 9.2. The PetTracker application: Entry Screen (left, middle) and Pet Listing Screen (right).

[View full size image]

- 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 9.2, middle).
- When you display the Pet List screen, we query for all pets and use a `Cursor` to programmatically build a TableLayout on-the-fly (Figure 9.2, 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 here to map the database results to the individual user interface components. If you decided you want to use a different widget to display your data, you would 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.

Authors' 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 Widgets Using Data Adapters

Ideally, you'd like to bind your data to user interface widgets 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`.

We included a project called "SuperPetTracker" on the CD accompanying this book (also available on the book Web site) that does this. It behaves much like the "PetTracker" sample application, except that it uses the `SimpleCursorAdapter` with `ListView` and an `ArrayAdapter` to handle`AutoCompleteTextView` features.

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.

As you remember from Chapter 7, "Designing Android User Interfaces with Layouts," the `ListView` container widget 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` which 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>
```

And 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` widgets 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 as well as the expected name and type columns appears in the query. This is required for the adapter and `ListView` to work properly.

Using a `ListView` (Figure 9.3, left) instead of a custom user interface allows us to take advantage of the `ListView` widget'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 9.3. The SuperPetTracker application: Pet Listing Screen `ListView` (left) with Delete feature (right).
[View full size image]

Now we reimplement the Delete functionality by listening for `onItemClick()` events and providing a Delete Confirmation dialog (Figure 9.3, right).

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

        final long deletePetId =  id;

        new AlertDialog.Builder(SuperPetList.this).setMessage(
          "Delete Pet Record?").setPositiveButton(
          "Delete", new DialogInterface.OnClickListener() {

              @Override
              public void onClick(DialogInterface dialog,int which) {
                  deletePet(deletePetId);
                  mCursor.requery();
        }}).show();
    }
});
```

You can see what this would look like on the screen in Figure 9.3.

Note that within the SuperPetTracker 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 will look like. You can find the built-in layout resources provided within your Android SDK directory under the directory **`tools\lib\res\default\layout`**.

Storing Nonprimitive Types (like Images) in the Database

Because SQLite is a single file, it makes little sense to try to store binary data within the database. Instead store the *location* of data, as a file path or a URI in the database and access it appropriately. We show an example of storing image URIs in the database in the next section.