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 `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.

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 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 identifiers. 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 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, 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 are 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, 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 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 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` 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.

```
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 10.1.

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.

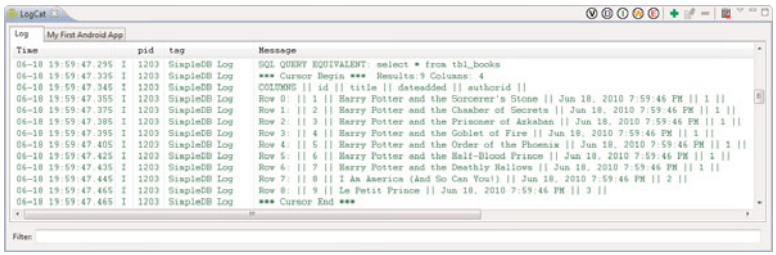

Figure 10.1 Sample log output for the **`logCursorInfo()`** method.

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 in the chapter, we called the `query()` method with only one parameter set to 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.

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 within a table by their fully qualified names. For example, the title column within 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 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

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 phone. 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 within 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. This source code for the PetTracker application is provided for download on the book website. We build upon this example in this and future chapters.

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()`, `onUpgrade()`, 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 "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 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 book website, 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 10.2, left).

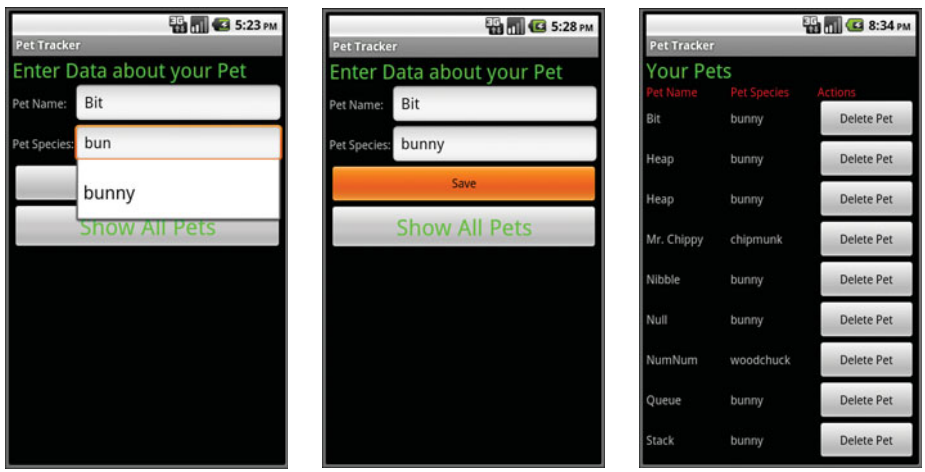

Figure 10.2 The PetTracker application: Entry Screen (left, middle) and Pet Listing Screen (right).

- When you save new records using the Pet Entry Form (Figure 10.2, 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 10.2, 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 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`.

We included a project called PetTracker2 on the book website 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.

> **Tip**
>
> The source code for subsequent upgrades to the PetTracker application (for example, PetTracker2, PetTracker3, and so on) is provided for download on the book website.

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.

Remember from Chapter 8, "Designing User Interfaces with Layouts," that the `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 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 10.3, 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.

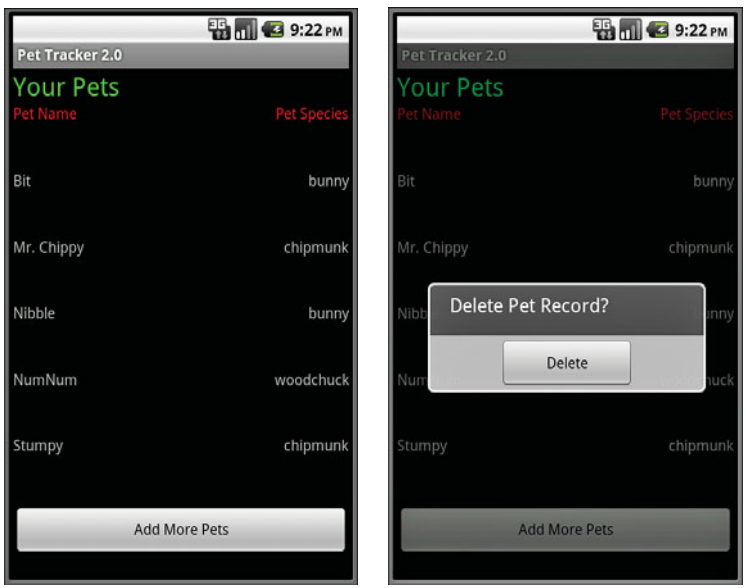

Figure 10.3 The PetTracker2 application: Pet Listing Screen **`ListView`** (left) with Delete feature (right).

Now we re-implement the Delete functionality by listening for `onItemClick()` events and providing a Delete Confirmation dialog (Figure 10.3, right):

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

        final long deletePetId =  id;

        new AlertDialog.Builder(PetTrackerListActivity.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 10.3.

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.

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 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 chapter.

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. With these skills, you are well on your way to leveraging one of the more powerful and unique features of Android.

Your application can store data using the following mechanisms:

- Lightweight application preferences (Activity-level and Application-wide)
- Android file system file and directory support with XML file format support
- Application-specific SQLite databases for structured storage

You learned how to design persistent data-access mechanisms within your Android application, and you understand how to bind data from various sources to user interface controls, such as `ListView` objects.

References and More Information

SQLite website:
http://www.sqlite.org/index.html
SQLzoo.net:
http://sqlzoo.net/