57. An Overview of Android SQLite Databases

Mobile applications that do not need to store at least some amount of persistent data are few and far between. The use of databases is an essential aspect of most applications, ranging from applications that are almost entirely data driven, to those that simply need to store small amounts of data such as the prevailing score of a game.

The importance of persistent data storage becomes even more evident when taking into consideration the somewhat transient lifecycle of the typical Android application. With the ever-present risk that the Android runtime system will terminate an application component to free up resources, a comprehensive data storage strategy to avoid data loss is a key factor in the design and implementation of any application development strategy.

This chapter will provide an overview of the SQLite database management system bundled with the Android operating system, together with an outline of the Android SDK classes that are provided to facilitate persistent SQLite based database storage from within an Android application. Before delving into the specifics of SQLite in the context of Android development, however, a brief overview of databases and SQL will be covered.

57.1 Understanding Database Tables

Database *Tables* provide the most basic level of data structure in a database. Each database can contain multiple tables and each table is designed to hold information of a specific type. For example, a database may contain a *customer* table that contains the name, address and telephone number for each of the customers of a particular business. The same database may also include a *products* table used to store the product descriptions with associated product codes for the items sold by the business.

Each table in a database is assigned a name that must be unique within that particular database. A table name, once assigned to a table in one database, may not be used for another table except within the context of another database.

57.2 Introducing Database Schema

Database Schemas define the characteristics of the data stored in a database table. For example, the table schema for a customer database table might define that the customer name is a string of no more than 20 characters in length, and that the customer phone number is a numerical data field of a certain format.

Schemas are also used to define the structure of entire databases and the relationship between the various tables contained in each database.

57.3 Columns and Data Types

It is helpful at this stage to begin to view a database table as being similar to a spreadsheet where data is stored in rows and columns.

Each column represents a data field in the corresponding table. For example, the name, address and telephone data fields of a table are all *columns*.

Each column, in turn, is defined to contain a certain type of data. A column designed to store numbers

would, therefore, be defined as containing numerical data.

57.4 Database Rows

Each new record that is saved to a table is stored in a row. Each row, in turn, consists of the columns of data associated with the saved record.

Once again, consider the spreadsheet analogy described earlier in this chapter. Each entry in a customer table is equivalent to a row in a spreadsheet and each column contains the data for each customer (name, address, telephone etc). When a new customer is added to the table, a new row is created and the data for that customer stored in the corresponding columns of the new row.

Rows are also sometimes referred to as *records* or *entries* and these terms can generally be used interchangeably.

57.5 Introducing Primary Keys

Each database table should contain one or more columns that can be used to identify each row in the table uniquely. This is known in database terminology as the *Primary Key*. For example, a table may use a bank account number column as the primary key. Alternatively, a customer table may use the customer's social security number as the primary key.

Primary keys allow the database management system to identify a specific row in a table uniquely. Without a primary key it would not be possible to retrieve or delete a specific row in a table because there can be no certainty that the correct row has been selected. For example, suppose a table existed where the customer's last name had been defined as the primary key. Imagine then the problem that might arise if more than one customer named "Smith" were recorded in the database. Without some guaranteed way to identify a specific row uniquely, it would be impossible to ensure the correct data was being accessed at any given time.

Primary keys can comprise a single column or multiple columns in a table. To qualify as a single column primary key, no two rows can contain matching primary key values. When using multiple columns to construct a primary key, individual column values do not need to be unique, but all the columns' values combined together must be unique.

57.6 What is SQLite?

SQLite is an embedded, relational database management system (RDBMS). Most relational databases (Oracle, SQL Server and MySQL being prime examples) are standalone server processes that run independently, and in cooperation with, applications that require database access. SQLite is referred to as *embedded* because it is provided in the form of a library that is linked into applications. As such, there is no standalone database server running in the background. All database operations are handled internally within the application through calls to functions contained in the SQLite library.

The developers of SQLite have placed the technology into the public domain with the result that it is now a widely deployed database solution.

SQLite is written in the C programming language and as such, the Android SDK provides a Java based “wrapper” around the underlying database interface. This essentially consists of a set of classes that may be utilized within the Java code of an application to create and manage SQLite based databases.

For additional information about SQLite refer to http://www.sqlite.org.

57.7 Structured Query Language (SQL)

Data is accessed in SQLite databases using a high-level language known as Structured Query Language. This is usually abbreviated to SQL and pronounced *sequel*. SQL is a standard language used by most relational database management systems. SQLite conforms mostly to the SQL-92 standard.

SQL is essentially a very simple and easy to use language designed specifically to enable the reading and writing of database data. Because SQL contains a small set of keywords, it can be learned quickly. In addition, SQL syntax is more or less identical between most DBMS implementations, so having learned SQL for one system, it is likely that your skills will transfer to other database management systems.

While some basic SQL statements will be used within this chapter, a detailed overview of SQL is beyond the scope of this book. There are, however, many other resources that provide a far better overview of SQL than we could ever hope to provide in a single chapter here.

57.8 Trying SQLite on an Android Virtual Device (AVD)

For readers unfamiliar with databases in general and SQLite in particular, diving right into creating an Android application that uses SQLite may seem a little intimidating. Fortunately, Android is shipped with SQLite pre-installed, including an interactive environment for issuing SQL commands from within an *adb shell* session connected to a running Android AVD emulator instance. This is both a useful way to learn about SQLite and SQL, and also an invaluable tool for identifying problems with databases created by applications running in an emulator.

To launch an interactive SQLite session, begin by running an AVD session. This can be achieved from within Android Studio by launching the Android Virtual Device Manager (*Tools -> Android -> AVD Manager*), selecting a previously configured AVD and clicking on the start button.

Once the AVD is up and running, open a Terminal or Command-Prompt window and connect to the emulator using the *adb* command-line tool as follows (note that the –e flag directs the tool to look for an emulator with which to connect, rather than a physical device):

```
adb -e shell
```

Once connected, the shell environment will provide a command prompt at which commands may be entered:

```
root@android:/ #
```

Data stored in SQLite databases are actually stored in database files on the file system of the Android device on which the application is running. By default, the file system path for these database files is as follows:

```
/data/data/<package name>/databases/<database filename>.db
```

For example, if an application with the package name *com.example.MyDBApp* creates a database named *mydatabase.db*, the path to the file on the device would read as follows:

```
/data/data/com.example.MyDBApp/databases/mydatabase.db
```

For the purposes of this exercise, therefore, change directory to /data/data within the adb shell and

create a sub-directory hierarchy suitable for some SQLite experimentation:

```
cd /data/data
mkdir com.example.dbexample
cd com.example.dbexample
mkdir databases
cd databases
```

With a suitable location created for the database file, launch the interactive SQLite tool as follows:

```
root@android:/data/data/databases # sqlite3 ./mydatabase.db
sqlite3 ./mydatabase.db
SQLite version 3.8.10.2 2015-05-20 18:17:19
Enter ".help" for usage hints.
sqlite>
```

At the *sqlite>* prompt, commands may be entered to perform tasks such as creating tables and inserting and retrieving data. For example, to create a new table in our database with fields to hold ID, name, address and phone number fields the following statement is required:

```
create table contacts (_id integer primary key autoincrement, name text,
address text, phone text);
```

Note that each row in a table should have a *primary key* that is unique to that row. In the above example, we have designated the ID field as the primary key, declared it as being of type *integer* and asked SQLite to increment the number automatically each time a row is added. This is a common way to make sure that each row has a unique primary key. On most other platforms, the choice of name for the primary key is arbitrary. In the case of Android, however, it is essential that the key be named *_id* in order for the database to be fully accessible using all of the Android database related classes. The remaining fields are each declared as being of type *text*.

To list the tables in the currently selected database, use the *.tables* statement:

```
sqlite> .tables
contacts
```

To insert records into the table:

```
sqlite> insert into contacts (name, address, phone) values ("Bill Smith",
"123 Main Street, California", "123-555-2323");
sqlite> insert into contacts (name, address, phone) values ("Mike Parks",
"10 Upping Street, Idaho", "444-444-1212");
```

To retrieve all rows from a table:

```
sqlite> select * from contacts;
1|Bill Smith|123 Main Street, California|123-555-2323
2|Mike Parks|10 Upping Street, Idaho|444-444-1212
```

To extract a row that meets specific criteria:

```
sqlite> select * from contacts where name="Mike Parks";
2|Mike Parks|10 Upping Street, Idaho|444-444-1212
```

To exit from the sqlite3 interactive environment:

```
sqlite> .exit
```

When running an Android application in the emulator environment, any database files will be created on the file system of the emulator using the previously discussed path convention. This has the advantage that you can connect with adb, navigate to the location of the database file, load it into the

sqlite3 interactive tool and perform tasks on the data to identify possible problems occurring in the application code.

It is also important to note that, while it is possible to connect with an adb shell to a physical Android device, the shell is not granted sufficient privileges by default to create and manage SQLite databases. Debugging of database problems is, therefore, best performed using an AVD session.

57.9 Android SQLite Java Classes

SQLite is, as previously mentioned, written in the C programming language while Android applications are primarily developed using Java. To bridge this "language gap", the Android SDK includes a set of classes that provide a Java layer on top of the SQLite database management system. The remainder of this chapter will provide a basic overview of each of the major classes within this category. More details on each class can be found in the online Android documentation.

57.9.1 Cursor

A class provided specifically to provide access to the results of a database query. For example, a SQL SELECT operation performed on a database will potentially return multiple matching rows from the database. A Cursor instance can be used to step through these results, which may then be accessed from within the application code using a variety of methods. Some key methods of this class are as follows:

- **close()** – Releases all resources used by the cursor and closes it.
- **getCount()** – Returns the number of rows contained within the result set.
- **moveToFirst()** – Moves to the first row within the result set.
- **moveToLast()** – Moves to the last row in the result set.
- **moveToNext()** – Moves to the next row in the result set.
- **move()** – Moves by a specified offset from the current position in the result set.
- **get<*type*>()** – Returns the value of the specified <*type*> contained at the specified column index of the row at the current cursor position (variations consist of *getString()*, *getInt()*, *getShort()*, *getFloat()* and *getDouble()*).

57.9.2 SQLiteDatabase

This class provides the primary interface between the application code and underlying SQLite databases including the ability to create, delete and perform SQL based operations on databases. Some key methods of this class are as follows:

- **insert()** – Inserts a new row into a database table.
- **delete()** – Deletes rows from a database table.
- **query()** – Performs a specified database query and returns matching results via a Cursor object.
- **execSQL()** – Executes a single SQL statement that does not return result data.
- **rawQuery()** – Executes a SQL query statement and returns matching results in the form of a Cursor object.

57.9.3 SQLiteOpenHelper

A helper class designed to make it easier to create and update databases. This class must be subclassed within the code of the application seeking database access and the following callback

methods implemented within that subclass:

- **onCreate()** – Called when the database is created for the first time. This method is passed the SQLiteDatabase object as an argument for the newly created database. This is the ideal location to initialize the database in terms of creating a table and inserting any initial data rows.
- **onUpgrade()** – Called in the event that the application code contains a more recent database version number reference. This is typically used when an application is updated on the device and requires that the database schema also be updated to handle storage of additional data.

In addition to the above mandatory callback methods, the *onOpen()* method, called when the database is opened, may also be implemented within the subclass.

The constructor for the subclass must also be implemented to call the super class, passing through the application context, the name of the database and the database version.

Notable methods of the SQLiteOpenHelper class include:

- **getWritableDatabase()** – Opens or creates a database for reading and writing. Returns a reference to the database in the form of a SQLiteDatabase object.
- **getReadableDatabase()** – Creates or opens a database for reading only. Returns a reference to the database in the form of a SQLiteDatabase object.
- **close**() – Closes the database.

57.9.4 ContentValues

ContentValues is a convenience class that allows key/value pairs to be declared consisting of table column identifiers and the values to be stored in each column. This class is of particular use when inserting or updating entries in a database table.

57.10 Summary

SQLite is a lightweight, embedded relational database management system that is included as part of the Android framework and provides a mechanism for implementing organized persistent data storage for Android applications. In addition to the SQLite database, the Android framework also includes a range of Java classes that may be used to create and manage SQLite based databases and tables.

The goal of this chapter was to provide an overview of databases in general and SQLite in particular within the context of Android application development. The next chapters will work through the creation of an example application intended to put this theory into practice in the form of a step-by-step tutorial. Since the user interface for the example application will require a forms based layout, the first chapter, entitled *An Android TableLayout and TableRow Tutorial*, will detour slightly from the core topic by introducing the basics of the TableLayout and TableRow views.

58. An Android TableLayout and TableRow Tutorial

When the work began on the next chapter of this book (*An Android SQLite Database Tutorial*) it was originally intended that it would include the steps to design the user interface layout for the database example application. It quickly became evident, however, that the best way to implement the user interface was to make use of the Android TableLayout and TableRow views and that this topic area deserved a self-contained chapter. As a result, this chapter will focus solely on the user interface design of the database application completed in the next chapter, and in doing so, take some time to introduce the basic concepts of table layouts in Android Studio.

58.1 The TableLayout and TableRow Layout Views

The purpose of the TableLayout container view is to allow user interface elements to be organized on the screen in a table format consisting of rows and columns. Each row within a TableLayout is occupied by a TableRow instance, which, in turn, is divided into cells, with each cell containing a single child view (which may itself be a container with multiple view children).

The number of columns in a table is dictated by the row with the most columns and, by default, the width of each column is defined by the widest cell in that column. Columns may be configured to be shrinkable or stretchable (or both) such that they change in size relative to the parent TableLayout. In addition, a single cell may be configured to span multiple columns.

Consider the user interface layout shown in Figure 58-1:

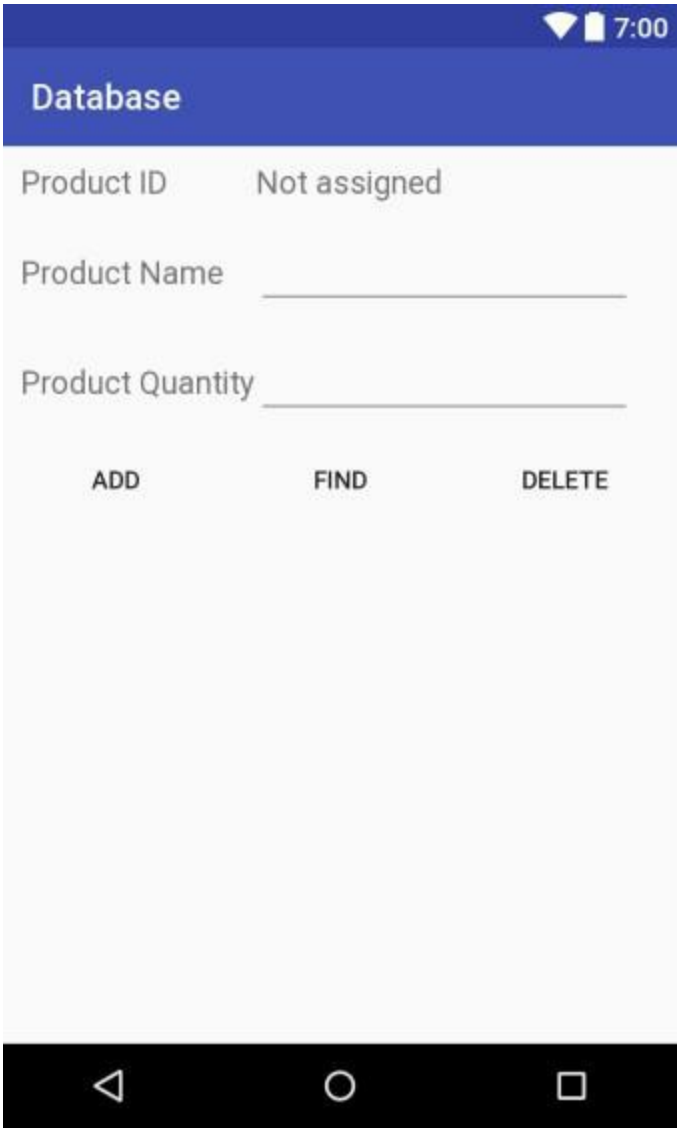

Figure 58-1

From the visual appearance of the layout, it is difficult to identify the TableLayout structure used to design the interface. The hierarchical tree illustrated in Figure 58-2, however, makes the structure a little easier to understand:

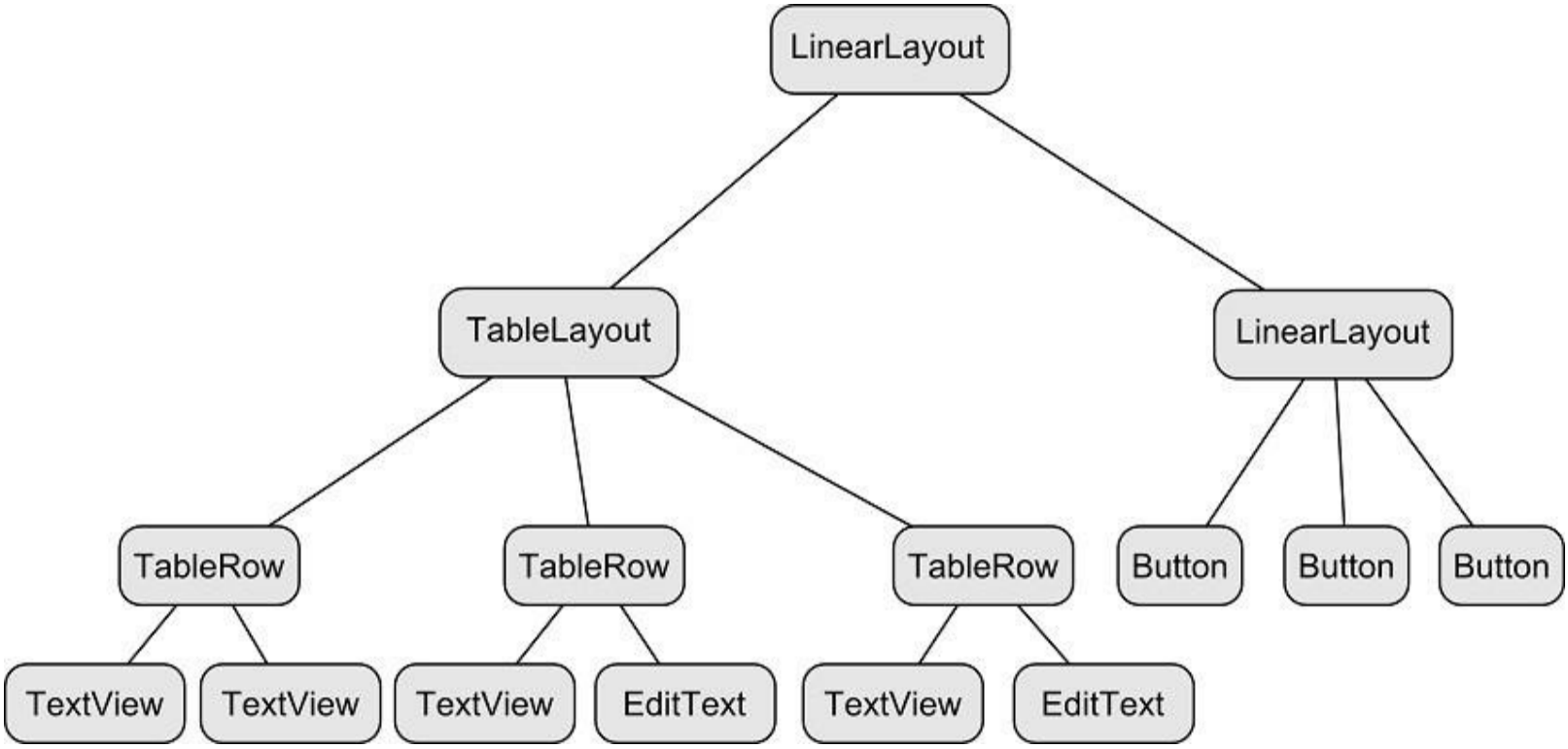

Figure 58-2

Clearly, the layout consists of a parent LinearLayout view with TableLayout and LinearLayout children. The TableLayout contains three TableRow children representing three rows in the table. The TableRows contain two child views, with each child representing the contents of a column cell. The LinearLayout child view contains three Button children.

The layout shown in Figure 58-2 is the exact layout that is required for the database example that will be completed in the next chapter. The remainder of this chapter, therefore, will be used to work step by step through the design of this user interface using the Android Studio Layout Editor tool.

58.2 Creating the Database Project

Start Android Studio and create a new project, entering *Database* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of an Empty Activity named *DatabaseActivity* with a corresponding layout file named *activity_database*.

58.3 Adding the TableLayout to the User Interface

Locate the *activity_database.xml* file in the Project tool window (*app -> res -> layout*) and double-click on it to load it into the Layout Editor tool. By default, Android Studio has used a ConstraintLayout as the root layout element in the user interface. This needs to be replaced by a vertically oriented LinearLayout. With the Layout Editor tool in Text mode, replace the XML with the following:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
```

```
        android:layout_height="match_parent"
        xmlns:android="http://schemas.android.com/apk/res/android">



    </LinearLayout>
```

Switch to Design mode and, referring to the Layouts category of the Palette, drag and drop a TableLayout view so that it is positioned at the top of the LinearLayout canvas area. With the LinearLayout component selected, use the Properties tool window to set the layout_height property to *wrap_content*.

Once these initial steps are complete, the Component Tree for the layout should resemble that shown in Figure 58-3.

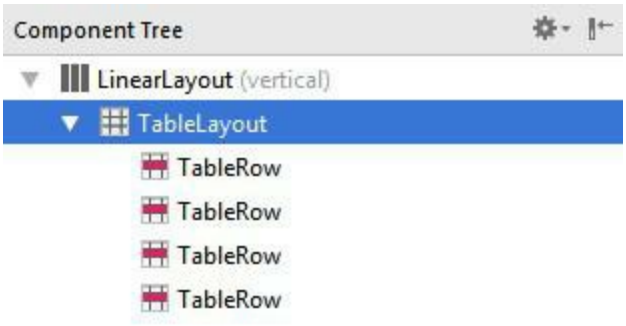

Figure 58-3

Clearly, Android Studio has automatically added four TableRow instances to the TableLayout. Since only three rows are required for this example, select and delete the fourth TableRow instance. Additional rows may be added to the TableLayout at any time by dragging the TableRow object from the palette and dropping it onto the TableLayout entry in the Component Tree tool window.

58.4 Configuring the TableRows

From within the *Widgets* section of the palette, drag and drop two TextView objects onto the uppermost TableRow entry in the Component Tree (Figure 58-4):

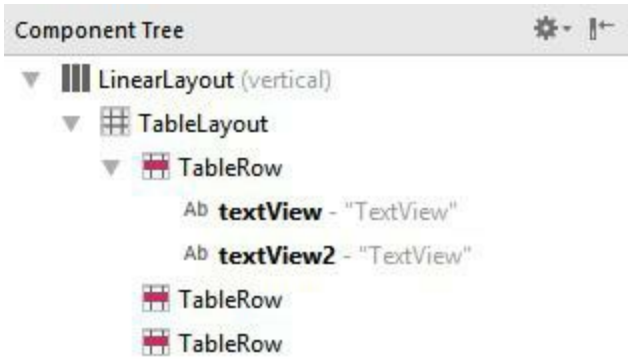

Figure 58-4

Select the left most TextView within the screen layout and, in the Properties tool window, change the *text* property to "Product ID". Repeat this step for the right most TextView, this time changing the text to "Not assigned" and specifying an *ID* value of *productID*.

Drag and drop another TextView widget onto the second TableRow entry in the Component Tree and change the text on the view to read "Product Name". Locate the Plain Text object in the palette and drag and drop it so that it is positioned beneath the Product Name TextView within the Component

Tree as outlined in Figure 58-5. With the TextView selected, change the inputType property from textPersonName to None, delete the "Name" string from the text property and set the ID to *productName*.

Figure 58-5

Drag and drop another TextView and a Number (Decimal) Text Field onto the third TableRow so that the TextView is positioned above the Text Field in the hierarchy. Change the text on the TextView to *Product Quantity* and the ID of the Text Field object to *productQuantity*.

Click and drag to select all of the widgets in the layout as shown in Figure 58-6 below, and use the Properties tool window to set the textSize property on all of the objects to 18sp:

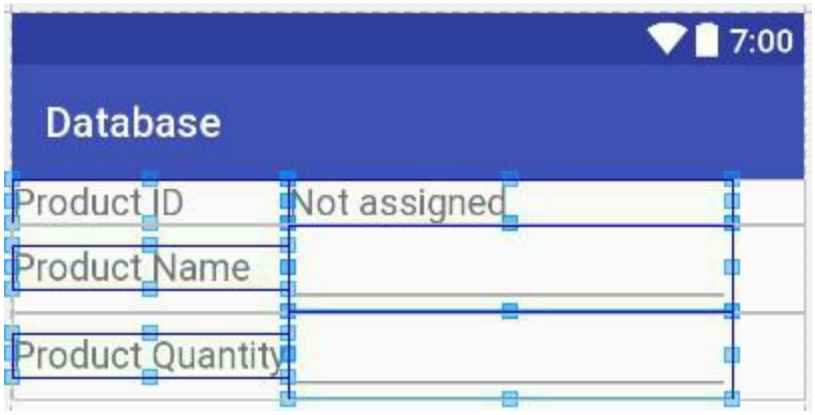

Figure 58-6

Before proceeding, be sure to extract all of the text properties added in the above steps to string resources.

58.5 Adding the Button Bar to the Layout

The next step is to add a LinearLayout (Horizontal) view to the parent LinearLayout view, positioned immediately below the TableLayout view. Begin by clicking on the small arrow to the left of the TableLayout entry in the Component Tree so that the TableRows are folded away from view. Drag a *LinearLayout (Horizontal)* instance from the *Layouts* section of the Layout Editor palette, drop it immediately beneath the TableLayout entry in the Component Tree panel and change the layout_height property to *wrap_content*:

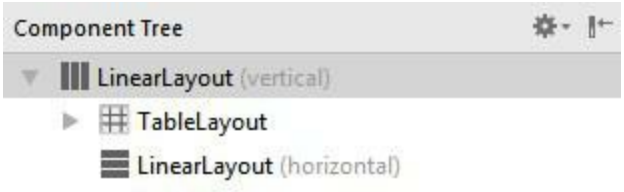

Figure 58-7

Drag and drop three Button objects onto the new LinearLayout and assign string resources for each button that read "Add", "Find" and "Delete" respectively. Buttons in this type of button bar arrangement should generally be displayed with a borderless style. For each button, use the Properties tool window to change the style setting to *Widget.AppCompat.Button.Borderless*.

With the new horizontal Linear Layout view selected in the Component Tree, switch the Properties panel to list all properties and change layout_gravity property to *center* (Figure 58-8) so that the buttons are centered horizontally within the display:

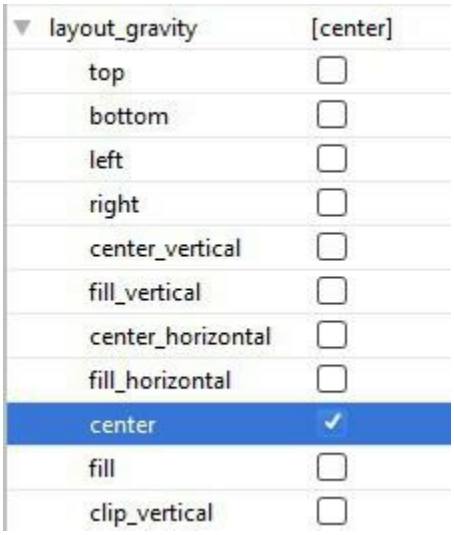

Figure 58-8

Before proceeding, check the hierarchy of the layout in the Component Tree panel, taking extra care to ensure the view ID names match those in the following figure:

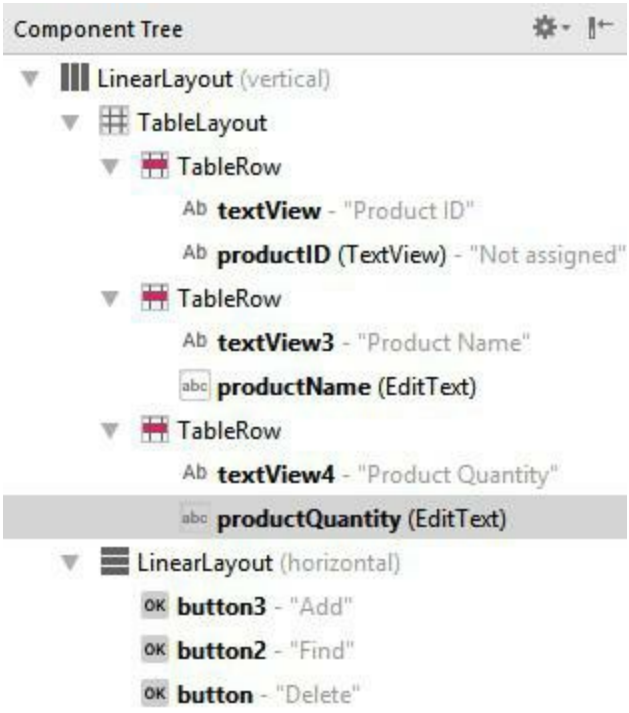

Figure 58-9

58.6 **Adjusting the Layout Margins**

All that remains is to adjust some of the layout settings. Begin by clicking on the first TableRow entry in the Component Tree panel so that it is selected. Hold down the Ctrl-key on the keyboard and click in the second and third TableRows so that all three items are selected. In the Properties panel, list all properties, locate the *layout_margins* properties category and, once located, change all the settings to 10dp as shown in Figure 58-10:

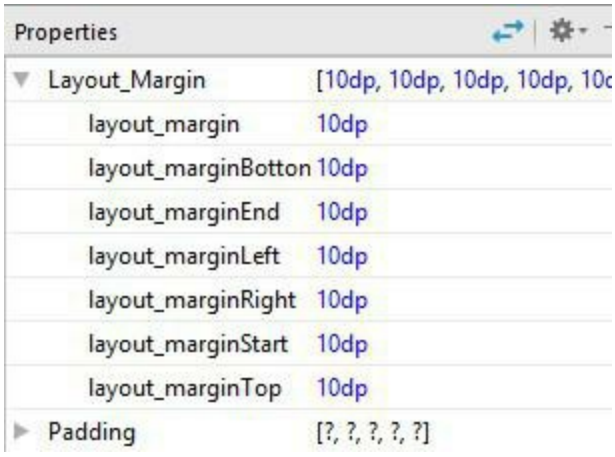

Figure 58-10

With margins set on all three TableRows, the user interface should appear as illustrated in Figure 58-1.

58.7 **Summary**

The Android TableLayout container view provides a way to arrange view components in a row and column configuration. While the TableLayout view provides the overall container, each row and the cells contained therein are implemented via instances of the TableRow view. In this chapter, a user interface has been designed in Android Studio using the TableLayout and TableRow containers. The next chapter will add the functionality behind this user interface to implement the SQLite database capabilities.

59. An Android SQLite Database Tutorial

The chapter entitled *An Overview of Android SQLite Databases* covered the basic principles of integrating relational database storage into Android applications using the SQLite database management system. The previous chapter took a minor detour into the territory of designing TableLayouts within the Android Studio Layout Editor tool, in the course of which, the user interface for an example database application was created. In this chapter, work on the *Database* application project will be continued with the ultimate objective of completing the database example.

59.1 About the Database Example

As is probably evident from the user interface layout designed in the preceding chapter, the example project is a simple data entry and retrieval application designed to allow the user to add, query and delete database entries. The idea behind this application is to allow the tracking of product inventory.

The name of the database will be *productID.db* which, in turn, will contain a single table named *products*. Each record in the database table will contain a unique product ID, a product description and the quantity of that product item currently in stock, corresponding to column names of "productid", "productname" and "productquantity", respectively. The productid column will act as the primary key and will be automatically assigned and incremented by the database management system.

The database schema for the *products* table is outlined in Table 59-1:

Column	Data Type
productid	Integer / Primary Key / Auto Increment
productname	Text
productquantity	Integer

Table 59-1

59.2 Creating the Data Model

Once completed, the application will consist of an activity and a database handler class. The database handler will be a subclass of SQLiteOpenHelper and will provide an abstract layer between the underlying SQLite database and the activity class, with the activity calling on the database handler to interact with the database (adding, removing and querying database entries). In order to implement this interaction in a structured way, a third class will need to be implemented to hold the database entry data as it is passed between the activity and the handler. This is actually a very simple class capable of holding product ID, product name and product quantity values, together with getter and setter methods for accessing these values. Instances of this class can then be created within the activity and database handler and passed back and forth as needed. Essentially, this class can be thought of as representing the database model.

Within Android Studio, navigate within the Project tool window to *app -> java* and right-click on the package name. From the popup menu, choose the *New -> Java Class* option and, in the *Create New Class* dialog, name the class *Product* before clicking on the *OK* button.

Once created the *Product.java* source file will automatically load into the Android Studio editor. Once loaded, modify the code to add the appropriate data members and methods:

```
package com.ebookfrenzy.database;

public class Product {

    private int _id;
    private String _productname;
    private int _quantity;

    public Product() {

    }

    public Product(int id, String productname, int quantity) {
        this._id = id;
        this._productname = productname;
        this._quantity = quantity;
    }

    public Product(String productname, int quantity) {
        this._productname = productname;
        this._quantity = quantity;
    }

    public void setID(int id) {
        this._id = id;
    }

    public int getID() {
        return this._id;
    }

    public void setProductName(String productname) {
        this._productname = productname;
    }

    public String getProductName() {
        return this._productname;
    }

    public void setQuantity(int quantity) {
        this._quantity = quantity;
    }

    public int getQuantity() {
        return this._quantity;
    }
}
```

The completed class contains private data members for the internal storage of data columns from database entries and a set of methods to get and set those values.

59.3 Implementing the Data Handler

The data handler will be implemented by subclassing from the Android SQLiteOpenHelper class and, as outlined in *An Overview of Android SQLite Databases*, adding the constructor, *onCreate()* and *onUpgrade()* methods. Since the handler will be required to add, query and delete data on behalf of the activity component, corresponding methods will also need to be added to the class.

Begin by adding a second new class to the project to act as the handler, this time named *MyDBHandler*. Once the new class has been created, modify it so that it reads as follows:

```
package com.ebookfrenzy.database;

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

public class MyDBHandler extends SQLiteOpenHelper {

    @Override
    public void onCreate(SQLiteDatabase db) {

    }

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

    }

}
```

Having now pre-populated the source file with template *onCreate()* and *onUpgrade()* methods, the next task is to add a constructor method. Modify the code to declare constants for the database name, table name, table columns and database version and to add the constructor method as follows:

```
package com.ebookfrenzy.database;

import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.content.Context;
import android.content.ContentValues;
import android.database.Cursor;

public class MyDBHandler extends SQLiteOpenHelper {

    private static final int DATABASE_VERSION = 1;
    private static final String DATABASE_NAME = "productDB.db";
    public static final String TABLE_PRODUCTS = "products";

    public static final String COLUMN_ID = "_id";
    public static final String COLUMN_PRODUCTNAME = "productname";
    public static final String COLUMN_QUANTITY = "quantity";

    public MyDBHandler(Context context, String name,
      SQLiteDatabase.CursorFactory factory, int version) {
        super(context, DATABASE_NAME, factory, DATABASE_VERSION);
```

```
    }

    @Override
    public void onCreate(SQLiteDatabase db) {

    }

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

    }

}
```

Next, the *onCreate()* method needs to be implemented so that the *products* table is created when the database is first initialized. This involves constructing a SQL CREATE statement containing instructions to create a new table with the appropriate columns and then passing that through to the *execSQL()* method of the SQLiteDatabase object passed as an argument to *onCreate()*:

```
@Override
public void onCreate(SQLiteDatabase db) {
       String CREATE_PRODUCTS_TABLE = "CREATE TABLE " +
             TABLE_PRODUCTS + "("
             + COLUMN_ID + " INTEGER PRIMARY KEY," +
                      COLUMN_PRODUCTNAME
             + " TEXT," + COLUMN_QUANTITY + " INTEGER" + ")";
      db.execSQL(CREATE_PRODUCTS_TABLE);
}
```

The *onUpgrade()* method is called when the handler is invoked with a greater database version number from the one previously used. The exact steps to be performed in this instance will be application specific, so for the purposes of this example, we will simply remove the old database and create a new one:

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion,
                              int newVersion) {
       db.execSQL("DROP TABLE IF EXISTS " + TABLE_PRODUCTS);
       onCreate(db);
}
```

All that now remains to be implemented in the *MyDBHandler.java* handler class are the methods to add, query and remove database table entries.

59.3.1 The Add Handler Method

The method to insert database records will be named *addProduct()* and will take as an argument an instance of our Product data model class. A ContentValues object will be created in the body of the method and primed with key-value pairs for the data columns extracted from the Product object. Next, a reference to the database will be obtained via a call to *getWritableDatabase()* followed by a call to the *insert()* method of the returned database object. Finally, once the insertion has been performed, the database needs to be closed:

```
public void addProduct(Product product) {
```

```
        ContentValues values = new ContentValues();
        values.put(COLUMN_PRODUCTNAME, product.getProductName());
        values.put(COLUMN_QUANTITY, product.getQuantity());

        SQLiteDatabase db = this.getWritableDatabase();

        db.insert(TABLE_PRODUCTS, null, values);
        db.close();
    }
```

59.3.2 The Query Handler Method

The method to query the database will be named *findProduct()* and will take as an argument a String object containing the name of the product to be located. Using this string, a SQL SELECT statement will be constructed to find all matching records in the table. For the purposes of this example, only the first match will then be returned, contained within a new instance of our Product data model class:

```
    public Product findProduct(String productname) {
        String query = "SELECT * FROM " + TABLE_PRODUCTS + " WHERE " +
COLUMN_PRODUCTNAME + " =  \"" + productname + "\"";

        SQLiteDatabase db = this.getWritableDatabase();

        Cursor cursor = db.rawQuery(query, null);

        Product product = new Product();

        if (cursor.moveToFirst()) {
            cursor.moveToFirst();
            product.setID(Integer.parseInt(cursor.getString(0)));
            product.setProductName(cursor.getString(1));
            product.setQuantity(Integer.parseInt(cursor.getString(2)));
            cursor.close();
        } else {
            product = null;
        }
        db.close();
        return product;
    }
```

59.3.3 The Delete Handler Method

The deletion method will be named *deleteProduct()* and will accept as an argument the entry to be deleted in the form of a Product object. The method will use a SQL SELECT statement to search for the entry based on the product name and, if located, delete it from the table. The success or otherwise of the deletion will be reflected in a Boolean return value:

```
    public boolean deleteProduct(String productname) {

        boolean result = false;

        String query = "SELECT * FROM " + TABLE_PRODUCTS + " WHERE " +
COLUMN_PRODUCTNAME + " =  \"" + productname + "\"";

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

```
        Cursor cursor = db.rawQuery(query, null);

        Product product = new Product();

        if (cursor.moveToFirst()) {
               product.setID(Integer.parseInt(cursor.getString(0)));
               db.delete(TABLE_PRODUCTS, COLUMN_ID + " = ?",
                     new String[] { String.valueOf(product.getID()) });
               cursor.close();
               result = true;
        }
        db.close();
        return result;
    }
```

59.4 Implementing the Activity Event Methods

The final task prior to testing the application is to wire up *onClick* event handlers on the three buttons in the user interface and to implement corresponding methods for those events. Locate and load the *activity_database.xml* file into the Layout Editor tool, switch to Text mode and locate and modify the three button elements to add *onClick* properties:

```
<Button
    android:text="@string/add"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/button3"
    android:layout_weight="1"
    style="@style/Widget.AppCompat.Button.Borderless"
    android:onClick="newProduct" />


<Button
    android:text="@string/find"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/button2"
    android:layout_weight="1"
    style="@style/Widget.AppCompat.Button.Borderless"
    android:onClick="lookupProduct" />


<Button
    android:text="@string/delete"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/button"
    android:layout_weight="1"
    style="@style/Widget.AppCompat.Button.Borderless"
    android:onClick="removeProduct" />
```

Load the *DatabaseActivity.java* source file into the editor and implement the code to identify the views in the user interface and to implement the three "onClick" target methods:

```
package com.ebookfrenzy.database;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
```

```
import android.widget.EditText;
import android.widget.TextView;

public class DatabaseActivity extends AppCompatActivity {

    TextView idView;
    EditText productBox;
    EditText quantityBox;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_database);

        idView = (TextView) findViewById(R.id.productID);
        productBox = (EditText) findViewById(R.id.productName);
        quantityBox =
                (EditText) findViewById(R.id.productQuantity);
    }

    public void newProduct (View view) {
        MyDBHandler dbHandler = new MyDBHandler(this, null, null, 1);

        int quantity =
                Integer.parseInt(quantityBox.getText().toString());

        Product product =
                new Product(productBox.getText().toString(),
                            quantity);

        dbHandler.addProduct(product);
        productBox.setText("");
        quantityBox.setText("");
    }

    public void lookupProduct (View view) {
        MyDBHandler dbHandler = new MyDBHandler(this, null, null, 1);

        Product product = dbHandler.findProduct(
              productBox.getText().toString());

        if (product != null) {
            idView.setText(String.valueOf(product.getID()));

            quantityBox.setText(
                        String.valueOf(product.getQuantity()));
        } else {
            idView.setText("No Match Found");
        }
    }

    public void removeProduct (View view) {
        MyDBHandler dbHandler = new MyDBHandler(this, null,
                null, 1);

        boolean result = dbHandler.deleteProduct(
```

```
                productBox.getText().toString());

        if (result)
        {
            idView.setText("Record Deleted");
            productBox.setText("");
            quantityBox.setText("");
        }
        else
            idView.setText("No Match Found");
    }
}
```

59.5 Testing the Application

With the coding changes completed, compile and run the application either in an AVD session or on a physical Android device. Once the application is running, enter a product name and quantity value into the user interface form and touch the *Add* button. Once the record has been added the text boxes will clear. Repeat these steps to add a second product to the database. Next, enter the name of one of the newly added products into the product name field and touch the *Find* button. The form should update with the product ID and quantity for the selected product. Touch the *Delete* button to delete the selected record. A subsequent search by product name should indicate that the record no longer exists.

59.6 Summary

The purpose of this chapter has been to work step by step through a practical application of SQLite based database storage in Android applications. As an exercise to develop your new database skill set further, consider extending the example to include the ability to update existing records in the database table.

60. Understanding Android Content Providers

The previous chapter worked through the creation of an example application designed to store data using a SQLite database. When implemented in this way, the data is private to the application and, as such, inaccessible to other applications running on the same device. While this may be the desired behavior for many types of application, situations will inevitably arise whereby the data stored on behalf of an application could be of benefit to other applications. A prime example of this is the data stored by the built-in Contacts application on an Android device. While the Contacts application is primarily responsible for the management of the user's address book details, this data is also made accessible to any other applications that might need access to this data. This sharing of data between Android applications is achieved through the implementation of *content providers*.

60.1 What is a Content Provider?

A content provider provides access to structured data between different Android applications. This data is exposed to applications either as tables of data (in much the same way as a SQLite database) or as a handle to a file. This essentially involves the implementation of a client/server arrangement whereby the application seeking access to the data is the client and the content provider is the server, performing actions and returning results on behalf of the client.

A successful content provider implementation involves a number of different elements, each of which will be covered in detail in the remainder of this chapter.

60.2 The Content Provider

A content provider is created as a subclass of the *android.content.ContentProvider* class. Typically, the application responsible for managing the data to be shared will implement a content provider to facilitate the sharing of that data with other applications.

The creation of a content provider involves the implementation of a set of methods to manage the data on behalf of other, client applications. These methods are as follows:

60.2.1 onCreate()

This method is called when the content provider is first created and should be used to perform any initialization tasks required by the content provider.

60.2.2 query()

This method will be called when a client requests that data be retrieved from the content provider. It is the responsibility of this method to identify the data to be retrieved (either single or multiple rows), perform the data extraction and return the results wrapped in a Cursor object.

60.2.3 insert()

This method is called when a new row needs to be inserted into the provider database. This method must identify the destination for the data, perform the insertion and return the full URI of the newly added row.

60.2.4 update()

The method called when existing rows need to be updated on behalf of the client. The method uses the arguments passed through to update the appropriate table rows and return the number of rows updated as a result of the operation.

60.2.5 delete()

Called when rows are to be deleted from a table. This method deletes the designated rows and returns a count of the number of rows deleted.

60.2.6 getType()

Returns the MIME type of the data stored by the content provider.

It is important when implementing these methods in a content provider to keep in mind that, with the exception of the *onCreate()* method, they can be called from many processes simultaneously and must, therefore, be thread safe.

Once a content provider has been implemented, the issue that then arises is how the provider is identified within the Android system. This is where the *content URI* comes into play.

60.3 The Content URI

An Android device will potentially contain a number of content providers. The system must, therefore, provide some way of identifying one provider from another. Similarly, a single content provider may provide access to multiple forms of content (typically in the form of database tables). Client applications, therefore, need a way to specify the underlying data for which access is required. This is achieved through the use of content URIs.

The content URI is essentially used to identify specific data within a specific content provider. The *Authority* section of the URI identifies the content provider and usually takes the form of the package name of the content provider. For example:

```
com.example.mydbapp.myprovider
```

A specific database table within the provider data structure may be referenced by appending the table name to the authority. For example, the following URI references a table named *products* within the content provider:

```
com.example.mydbapp.myprovider/products
```

Similarly, a specific row within the specified table may be referenced by appending the row ID to the URI. The following URI, for example, references the row in the products table in which the value stored in the _ID column equals 3:

```
com.example.mydbapp.myprovider/products/3
```

When implementing the insert, query, update and delete methods in the content provider, it will be the responsibility of these methods to identify whether the incoming URI is targeting a specific row in a table, or references multiple rows, and act accordingly. This can potentially be a complex task given that a URI can extend to multiple levels. This process can, however, be eased significantly by making use of the *UriMatcher* class as will be outlined in the next chapter.

60.4 The Content Resolver

Access to a content provider is achieved via a ContentResolver object. An application can obtain a

reference to its content resolver by making a call to the *getContentResolver()* method of the application context.

The content resolver object contains a set of methods that mirror those of the content provider (insert, query, delete etc.). The application simply makes calls to the methods, specifying the URI of the content on which the operation is to be performed. The content resolver and content provider objects then communicate to perform the requested task on behalf of the application.

60.5 The <provider> Manifest Element

In order for a content provider to be visible within an Android system, it must be declared within the Android manifest file for the application in which it resides. This is achieved using the <*provider*> element, which must contain the following items:

- **android:authority** – The full authority URI of the content provider. For example com.example.mydbapp.mydbapp.myprovider.
- **android:name** – The name of the class that implements the content provider. In most cases, this will use the same value as the authority.

Similarly, the <provider> element may be used to define the permissions that must be held by client applications in order to qualify for access to the underlying data. If no permissions are declared, the default behavior is for permission to be allowed for all applications.

Permissions can be set to cover the entire content provider, or limited to specific tables and records.

60.6 Summary

The data belonging to an application is typically private to the application and inaccessible to other applications. In situations where the data needs to be shared, it is necessary to set up a content provider. This chapter has covered the basic elements that combine to enable data sharing between applications, and outlined the concepts of the content provider, content URI and content resolver.

In the next chapter, the Android Studio Database example application created previously will be extended to make the underlying product data available via a content provider.

61. Implementing an Android Content Provider in Android Studio

As outlined in the previous chapter, content providers provide a mechanism through which the data stored by one Android application can be made accessible to other applications. Having provided a theoretical overview of content providers, this chapter will continue the coverage of content providers by extending the Database project created in the chapter entitled *An Android TableLayout and TableRow Tutorial* to implement content provider based access to the database.

61.1 Copying the Database Project

In order to keep the original Database project intact, we will make a backup copy of the project before modifying it to implement content provider support for the application. If the Database project is currently open within Android Studio, close it using the *File -> Close Project* menu option.

Using the file system explorer for your operating system type, navigate to the directory containing your Android Studio projects (typically this will be a folder named *AndroidStudioProjects* located in your home directory). Within this folder, copy the Database project folder to a new folder named *DatabaseOriginal*.

Within the Android Studio welcome screen, select the *Open an existing Android Studio project* option from the Quick Start list and navigate to and select the original *Database* project so that it loads into the main window.

61.2 Adding the Content Provider Package

The next step is to add a new package to the Database project into which the content provider class will be created. Add this new package by navigating within the Project tool window to *app -> java,* right-clicking on the *java* entry and selecting the *New -> Package* menu option. When the *Choose Destination Directory* dialog appears, select the *..\app\src\main\java* option from the *Directory Structure* panel and click on OK.

In the *New Package* dialog, enter the following package name into the name field before clicking on the *OK* button:

```
com.ebookfrenzy.database.provider
```

The new package should now be listed within the Project tool window as illustrated in Figure 61-1:

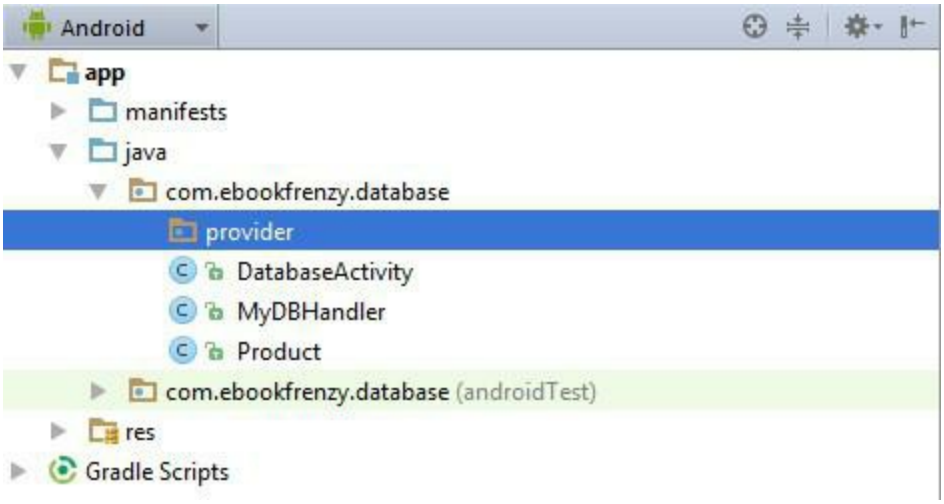

Figure 61-1

61.3 Creating the Content Provider Class

As discussed in *Understanding Android Content Providers*, content providers are created by subclassing the *android.content.ContentProvider* class. Consequently, the next step is to add a class to the new *provider* package to serve as the content provider for this application. Locate the new package in the Project tool window, right-click on it and select the *New -> Other -> Content Provider* menu option. In the *Configure Component* dialog, enter *MyContentProvider* into the *Class Name* field and the following into the *URI Authorities* field:

```
com.ebookfrenzy.database.provider.MyContentProvider
```

Make sure that the new content provider class is both exported and enabled before clicking on *Finish* to create the new class.

Once the new class has been created, the *MyContentProvider.java* file should be listed beneath the *provider* package in the Project tool window and automatically loaded into the editor where it should appear as outlined in the following listing:

```
package com.ebookfrenzy.database.provider;

import android.content.ContentProvider;
import android.content.ContentValues;
import android.database.Cursor;
import android.net.Uri;

public class MyContentProvider extends ContentProvider {
    public MyContentProvider() {
    }

    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        // Implement this to handle requests to delete one or more rows.
        throw new UnsupportedOperationException("Not yet implemented");
    }

    @Override
    public String getType(Uri uri) {
        // TODO: Implement this to handle requests for the MIME type of the
data
        // at the given URI.
```

```
        throw new UnsupportedOperationException("Not yet implemented");
    }

    @Override
    public Uri insert(Uri uri, ContentValues values) {
        // TODO: Implement this to handle requests to insert a new row.
        throw new UnsupportedOperationException("Not yet implemented");
    }

    @Override
    public boolean onCreate() {
        // TODO: Implement this to initialize your content provider on
startup.
        return false;
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
            String[] selectionArgs, String sortOrder) {
        // TODO: Implement this to handle query requests from clients.
        throw new UnsupportedOperationException("Not yet implemented");
    }

    @Override
    public int update(Uri uri, ContentValues values, String selection,
            String[] selectionArgs) {
        // TODO: Implement this to handle requests to update one or more
rows.
        throw new UnsupportedOperationException("Not yet implemented");
    }
}
```

As is evident from a quick review of the code in this file, Android Studio has already populated the class with stubs for each of the methods that a subclass of ContentProvider is required to implement. It will soon be necessary to begin implementing these methods, but first some constants relating to the provider's content authority and URI need to be declared.

61.4 Constructing the Authority and Content URI

As outlined in the previous chapter, all content providers must have associated with them an *authority* and a *content uri*. In practice, the authority is typically the full package name of the content provider class itself, in this case *com.ebookfrenzy.database.database.provider.MyContentProvider* as declared when the new Content Provider class was created in the previous section.

The content URI will vary depending on application requirements, but for the purposes of this example it will comprise the authority with the name of the database table appended at the end. Within the *MyContentProvider.java* file, make the following modifications:

```
package com.ebookfrenzy.database.provider;

import android.content.ContentProvider;
import android.content.ContentValues;
import android.database.Cursor;
import android.net.Uri;
import android.content.UriMatcher;
```

```
public class MyContentProvider extends ContentProvider {

    private static final String AUTHORITY =
      "com.ebookfrenzy.database.provider.MyContentProvider";
    private static final String PRODUCTS_TABLE = "products";
    public static final Uri CONTENT_URI =
            Uri.parse("content://" + AUTHORITY + "/" +
`                   PRODUCTS_TABLE);

    public MyContentProvider() {
    }
.
.
.
}
```

The above statements begin by creating a new String object named *AUTHORITY* and assigning the authority string to it. Similarly, a second String object named *PRODUCTS_TABLE* is created and initialized with the name of our database table (products).

Finally, these two string elements are combined, prefixed with *content://* and converted to a Uri object using the *parse()* method of the Uri class. The result is assigned to a variable named *CONTENT_URI*.

61.5 **Implementing URI Matching in the Content Provider**

When the methods of the content provider are called, they will be passed as an argument a URI indicating the data on which the operation is to be performed. This URI may take the form of a reference to a specific row in a specific table. It is also possible that the URI will be more general, for example specifying only the database table. It is the responsibility of each method to identify the *Uri type* and to act accordingly. This task can be eased considerably by making use of a UriMatcher instance. Once a UriMatcher instance has been created, it can be configured to return a specific integer value corresponding to the type of URI it detects when asked to do so. For the purposes of this tutorial, we will be configuring our UriMatcher instance to return a value of 1 when the URI references the entire products table, and a value of 2 when the URI references the ID of a specific row in the products table. Before working on creating the URIMatcher instance, we will first create two integer variables to represent the two URI types:

```
package com.ebookfrenzy.database.provider;

import android.content.ContentProvider;
import android.content.ContentValues;
import android.database.Cursor;
import android.net.Uri;
import android.content.UriMatcher;

public class MyContentProvider extends ContentProvider {

    private static final String AUTHORITY =
      "com.ebookfrenzy.database.provider.MyContentProvider";
    private static final String PRODUCTS_TABLE = "products";
    public static final Uri CONTENT_URI =
```

```
                Uri.parse("content://" + AUTHORITY + "/" +
                          PRODUCTS_TABLE);

        public static final int PRODUCTS = 1;
        public static final int PRODUCTS_ID = 2;
    .
    .
    }
```

With the Uri type variables declared, it is now time to add code to create a UriMatcher instance and configure it to return the appropriate variables:

```
    public class MyContentProvider extends ContentProvider {

        private static final String AUTHORITY =
          "com.ebookfrenzy.database.provider.MyContentProvider";
        private static final String PRODUCTS_TABLE = "products";
        public static final Uri CONTENT_URI =
                Uri.parse("content://" + AUTHORITY + "/" +
                          PRODUCTS_TABLE);

        public static final int PRODUCTS = 1;
        public static final int PRODUCTS_ID = 2;

        private static final UriMatcher sURIMatcher =
                                    new UriMatcher(UriMatcher.NO_MATCH);

        static {
                sURIMatcher.addURI(AUTHORITY, PRODUCTS_TABLE, PRODUCTS);
                sURIMatcher.addURI(AUTHORITY, PRODUCTS_TABLE + "/#",
                                                 PRODUCTS_ID);
        }
    .
    .
    }
```

The UriMatcher instance (named sURIMatcher) is now primed to return the value of PRODUCTS when just the products table is referenced in a URI, and PRODUCTS_ID when the URI includes the ID of a specific row in the table.

61.6 Implementing the Content Provider onCreate() Method

When the content provider class is created and initialized, a call will be made to the *onCreate()* method of the class. It is within this method that any initialization tasks for the class need to be performed. For the purposes of this example, all that needs to be performed is for an instance of the MyDBHandler class implemented in *An Android SQLite Database Tutorial* to be created. Once this instance has been created, it will need to be accessible from the other methods in the class, so a declaration for the database handler also needs to be declared, resulting in the following code changes to the *MyContentProvider.java* file:

```
    package com.ebookfrenzy.database.provider;

    import com.ebookfrenzy.database.MyDBHandler;

    import android.content.ContentProvider;
```

```
import android.content.ContentValues;
import android.database.Cursor;
import android.net.Uri;
import android.content.UriMatcher;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteQueryBuilder;
import android.text.TextUtils;

public class MyContentProvider extends ContentProvider {

        private MyDBHandler myDB;
.
.
.
        @Override
        public boolean onCreate() {
                myDB = new MyDBHandler(getContext(), null, null, 1);
               return false;
        }
}
```

61.7 Implementing the Content Provider insert() Method

When a client application or activity requests that data be inserted into the underlying database, the *insert()* method of the content provider class will be called. At this point, however, all that exists in the *MyContentProvider.java* file of the project is a stub method, which reads as follows:

```
@Override
public Uri insert(Uri uri, ContentValues values) {
    // TODO: Implement this to handle requests to insert a new row.
    throw new UnsupportedOperationException("Not yet implemented");
}
```

Passed as arguments to the method are a URI specifying the destination of the insertion and a ContentValues object containing the data to be inserted.

This method now needs to be modified to perform the following tasks:

- Use the sUriMatcher object to identify the URI type.
- Throw an exception if the URI is not valid.
- Obtain a reference to a writable instance of the underlying SQLite database.
- Perform a SQL insert operation to insert the data into the database table.
- Notify the corresponding content resolver that the database has been modified.
- Return the URI of the newly added table row.

Bringing these requirements together results in a modified *insert()* method, which reads as follows:

```
@Override
public Uri insert(Uri uri, ContentValues values) {

    int uriType = sURIMatcher.match(uri);

    SQLiteDatabase sqlDB = myDB.getWritableDatabase();

    long id = 0;
    switch (uriType) {
        case PRODUCTS:
```

```
                id = sqlDB.insert(MyDBHandler.TABLE_PRODUCTS,
                        null, values);
                break;
            default:
                throw new IllegalArgumentException("Unknown URI: "
                        + uri);
        }
        getContext().getContentResolver().notifyChange(uri, null);
        return Uri.parse(PRODUCTS_TABLE + "/" + id);
    }
```

61.8 Implementing the Content Provider query() Method

When a content provider is called upon to return data, the *query()* method of the provider class will be called. When called, this method is passed some or all of the following arguments:

- **URI** – The URI specifying the data source on which the query is to be performed. This can take the form of a general query with multiple results, or a specific query targeting the ID of a single table row.
- **Projection** – A row within a database table can comprise multiple columns of data. In the case of this application, for example, these correspond to the ID, product name and product quantity. The projection argument is simply a String array containing the name for each of the columns that is to be returned in the result data set.
- **Selection** – The "where" element of the selection to be performed as part of the query. This argument controls which rows are selected from the specified database. For example, if the query was required to select only products named "Cat Food" then the selection string passed to the query() method would read *productname = "Cat Food"*.
- **Selection Args** – Any additional arguments that need to be passed to the SQL query operation to perform the selection.
- **Sort Order** – The sort order for the selected rows.

When called, the *query()* method is required to perform the following operations:

- Use the sUriMatcher to identify the Uri type.
- Throw an exception if the URI is not valid.
- Construct a SQL query based on the criteria passed to the method. For convenience, the SQLiteQueryBuilder class can be used in construction of the query.
- Execute the query operation on the database.
- Notify the content resolver of the operation.
- Return a Cursor object containing the results of the query.

With these requirements in mind, the code for the *query()* method in the *MyContentProvider.java* file should now read as outlined in the following listing:

```
    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
            String[] selectionArgs, String sortOrder) {

        SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();
        queryBuilder.setTables(MyDBHandler.TABLE_PRODUCTS);

        int uriType = sURIMatcher.match(uri);
```

```
        switch (uriType) {
            case PRODUCTS_ID:
                queryBuilder.appendWhere(MyDBHandler.COLUMN_ID + "="
                        + uri.getLastPathSegment());
                break;
            case PRODUCTS:
                break;
            default:
                throw new IllegalArgumentException("Unknown URI");
        }

        Cursor cursor = queryBuilder.query(myDB.getReadableDatabase(),
                    projection, selection, selectionArgs, null, null,
                    sortOrder);
        cursor.setNotificationUri(getContext().getContentResolver(),
                    uri);
        return cursor;
    }
```

61.9 Implementing the Content Provider update() Method

The *update()* method of the content provider is called when changes are being requested to existing database table rows. The method is passed a URI with the new values in the form of a ContentValues object and the usual selection argument strings.

When called, the *update()* method would typically perform the following steps:

- Use the sUriMatcher to identify the URI type.
- Throw an exception if the URI is not valid.
- Obtain a reference to a writable instance of the underlying SQLite database.
- Perform the appropriate update operation on the database, depending on the selection criteria and the URI type.
- Notify the content resolver of the database change.
- Return a count of the number of rows that were changed as a result of the update operation.

A general-purpose *update()* method, and the one we will use for this project, would read as follows:

```
    @Override
    public int update(Uri uri, ContentValues values, String selection,
                String[] selectionArgs) {

                  int uriType = sURIMatcher.match(uri);
                  SQLiteDatabase sqlDB = myDB.getWritableDatabase();
                  int rowsUpdated = 0;

                  switch (uriType) {
                    case PRODUCTS:
                      rowsUpdated =
                          sqlDB.update(MyDBHandler.TABLE_PRODUCTS,
                          values,
                          selection,
                          selectionArgs);
                      break;
                    case PRODUCTS_ID:
                      String id = uri.getLastPathSegment();
                      if (TextUtils.isEmpty(selection)) {
```

```
                    rowsUpdated =
                      sqlDB.update(MyDBHandler.TABLE_PRODUCTS,
                        values,
                        MyDBHandler.COLUMN_ID + "=" + id,
                        null);
                    } else {
                      rowsUpdated =
                        sqlDB.update(MyDBHandler.TABLE_PRODUCTS,
                          values,
                          MyDBHandler.COLUMN_ID + "=" + id
                          + " and "
                          + selection,
                          selectionArgs);
                    }
                    break;
                  default:
                    throw new IllegalArgumentException("Unknown URI: "
                        + uri);
                  }
                 getContext().getContentResolver().notifyChange(uri,
                                                                null);
                return rowsUpdated;
    }
```

61.10 **Implementing the Content Provider delete() Method**

In common with a number of other content provider methods, the *delete()* method is passed a URI, a selection string and an optional set of selection arguments. A typical *delete()* method will also perform the following, and by now largely familiar, tasks when called:

- Use the sUriMatcher to identify the URI type.
- Throw an exception if the URI is not valid.
- Obtain a reference to a writable instance of the underlying SQLite database.
- Perform the appropriate delete operation on the database depending on the selection criteria and the Uri type.
- Notify the content resolver of the database change.
- Return the number of rows deleted as a result of the operation.

A typical *delete()* method is in many ways similar to the *update()* method and may be implemented as follows:

```
    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {
            int uriType = sURIMatcher.match(uri);
            SQLiteDatabase sqlDB = myDB.getWritableDatabase();
            int rowsDeleted = 0;

            switch (uriType) {
                case PRODUCTS:
                  rowsDeleted = sqlDB.delete(MyDBHandler.TABLE_PRODUCTS,
                   selection,
                    selectionArgs);
                    break;

                case PRODUCTS_ID:
                  String id = uri.getLastPathSegment();
```

```
            if (TextUtils.isEmpty(selection)) {
              rowsDeleted = sqlDB.delete(MyDBHandler.TABLE_PRODUCTS,
                              MyDBHandler.COLUMN_ID + "=" + id,
                      null);
            } else {
              rowsDeleted = sqlDB.delete(MyDBHandler.TABLE_PRODUCTS,
                              MyDBHandler.COLUMN_ID + "=" + id
                      + " and " + selection,
                      selectionArgs);
            }
            break;
          default:
            throw new IllegalArgumentException("Unknown URI: " +
                                                  uri);
          }
          getContext().getContentResolver().notifyChange(uri, null);
          return rowsDeleted;
    }
```

With these methods implemented, the content provider class, in terms of the requirements for this example at least, is complete. The next step is to make sure that the content provider is declared in the project manifest file so that it is visible to any content resolvers seeking access to it.

61.11 Declaring the Content Provider in the Manifest File

Unless a content provider is declared in the manifest file of the application to which it belongs, it will not be possible for a content resolver to locate and access it. As outlined, content providers are declared using the <provider> tag and the manifest entry must correctly reference the content provider authority and content URI.

For the purposes of this project, therefore, locate the *AndroidManifest.xml* file for the DatabaseProvider project within the Project tool window and double-click on it to load it into the editing panel. Within the editing panel, make sure that the content provider declaration has already been added by Android Studio when the MyContentProvider class was added to the project:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.ebookfrenzy.database" >

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".DatabaseActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <provider android:name="com.ebookfrenzy.database.MyContentProvider"
```

```
            android:authorities=
  "com.ebookfrenzy.database.provider.MyContentProvider"
            android:enabled="true"
            android:exported="true" >
        </provider>
    </application>

</manifest>
```

All that remains before testing the application is to modify the database handler class to use the content provider instead of directly accessing the database.

61.12 Modifying the Database Handler

When this application was originally created, it was designed to use a database handler to access the underlying database directly. Now that a content provider has been implemented, the database handler needs to be modified so that all database operations are performed using the content provider via a content resolver.

The first step is to modify the *MyDBHandler.java* class so that it obtains a reference to a ContentResolver instance. This can be achieved in the constructor method of the class:

```
package com.ebookfrenzy.database;

import com.ebookfrenzy.database.provider.MyContentProvider;

import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.content.Context;
import android.content.ContentValues;
import android.database.Cursor;
import android.content.ContentResolver;

public class MyDBHandler extends SQLiteOpenHelper {

    private ContentResolver myCR;

    private static final int DATABASE_VERSION = 1;
    private static final String DATABASE_NAME = "productDB.db";
    public static final String TABLE_PRODUCTS = "products";

    public static final String COLUMN_ID = "_id";
    public static final String COLUMN_PRODUCTNAME = "productname";
    public static final String COLUMN_QUANTITY = "quantity";

    public MyDBHandler(Context context, String name,
                       SQLiteDatabase.CursorFactory factory, int version) {
        super(context, DATABASE_NAME, factory, DATABASE_VERSION);
        myCR = context.getContentResolver();
    }
.
.
.
}
```

Next, the *addProduct()*, *findProduct()* and *removeProduct()* methods need to be rewritten to use the content resolver and content provider for data management purposes:

```
public void addProduct(Product product) {

        ContentValues values = new ContentValues();
        values.put(COLUMN_PRODUCTNAME, product.getProductName());
        values.put(COLUMN_QUANTITY, product.getQuantity());

        myCR.insert(MyContentProvider.CONTENT_URI, values);
}

public Product findProduct(String productname) {
        String[] projection = {COLUMN_ID,
           COLUMN_PRODUCTNAME, COLUMN_QUANTITY };

        String selection = "productname = \"" + productname + "\"";

        Cursor cursor = myCR.query(MyContentProvider.CONTENT_URI,
               projection, selection, null,
                null);

        Product product = new Product();

        if (cursor.moveToFirst()) {
               cursor.moveToFirst();
               product.setID(Integer.parseInt(cursor.getString(0)));
               product.setProductName(cursor.getString(1));
               product.setQuantity(
                       Integer.parseInt(cursor.getString(2)));
               cursor.close();
        } else {
               product = null;
        }
        return product;
}

public boolean deleteProduct(String productname) {

        boolean result = false;

        String selection = "productname = \"" + productname + "\"";

        int rowsDeleted = myCR.delete(MyContentProvider.CONTENT_URI,
                   selection, null);

        if (rowsDeleted > 0)
               result = true;

        return result;
}
```

With the database handler class updated to use a content resolver and content provider, the application is now ready to be tested. Compile and run the application and perform some operations to add, find and remove product entries. In terms of operation and functionality, the application should behave exactly as it did when directly accessing the database, except that it is now using the

content provider.

With the content provider now implemented and declared in the manifest file, any other applications can potentially access that data (since no permissions were declared, the default full access is in effect). The only information that the other applications need to know to gain access is the content URI and the names of the columns in the products table.

61.13 **Summary**

The goal of this chapter was to provide a more detailed overview of the exact steps involved in implementing an Android content provider with a particular emphasis on the structure and implementation of the query, insert, delete and update methods of the content provider class. Practical use of the content resolver class to access data in the content provider was also covered, and the Database project was modified to make use of both a content provider and content resolver.