10

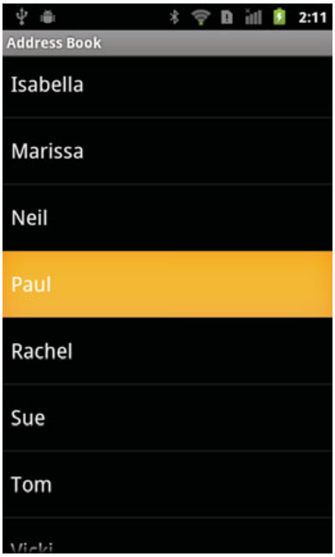
Address Book App

ListActivity, AdapterViews, Adapters, Multiple Activities, SQLite, GUI Styles, Menu Resources and MenuInflater

Objectives

In this chapter you'll:

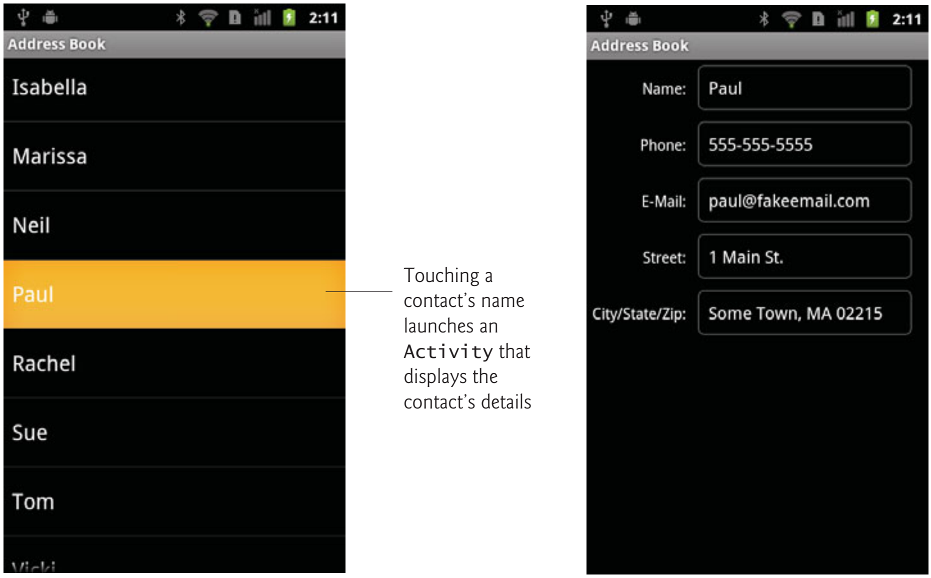
- Extend ListActivity to create an Activity that consists of a ListView by default.
- Create multiple Activity subclasses to represent the app's tasks and use explicit Intents to launch them.
- Create and open SQLite databases using a SQLiteOpenHelper, and insert, delete and query data in a SQLite database using a SQLiteDatabase object
- Use a SimpleCursorAdapter to bind database query results to a ListView's items.
- Use a Cursor to manipulate database query results.
- Use multithreading to perform database operations outside the GUI thread and maintain application responsiveness.
- Define styles containing common GUI attributes and values, then apply them to multiple GUI components.
- Create XML menu resources and inflate them with a MenuInflater.



Outline

10.1 Introduction

The **Address Book** app (Fig. 10.1) provides convenient access to stored contact information. On the main screen, the user can *scroll* through an alphabetical contact list and can view a contact's details by touching the contact's name. Touching the device's menu button while viewing a contact's details displays a menu containing **Edit Contact** and **Delete Contact** options (Fig. 10.2). If the user chooses to edit the contact, the app launches an `Activity` that shows the existing information in `EditTexts` (Fig. 10.2). If the user chooses to delete the contact, a dialog asks the user to confirm the delete operation (Fig. 10.3). Touching the device's menu button while viewing the contact list displays a menu con-



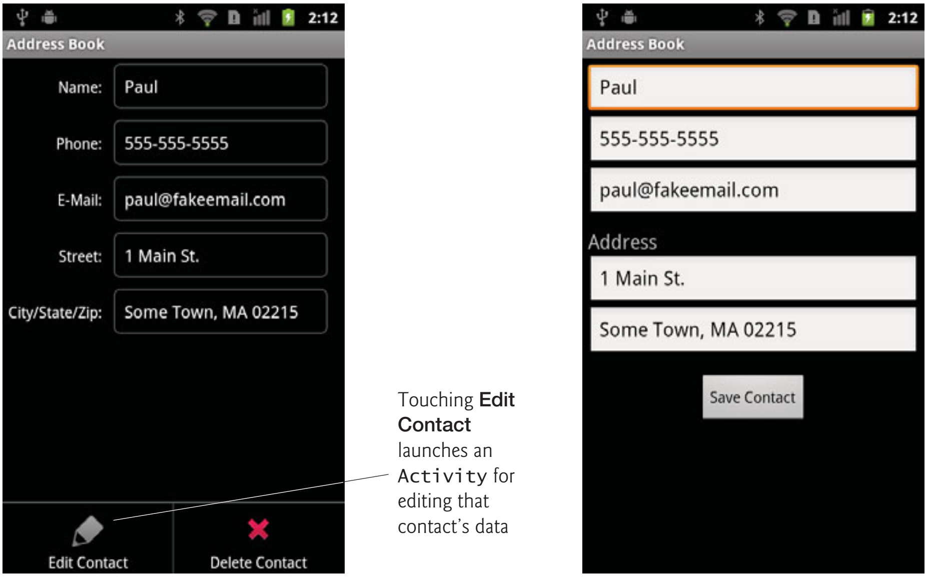
Isabella
Marissa
Neil
Paul
Rachel
Sue
Tom
Vicki

Name: Paul
Phone: 555-555-5555
E-Mail: paul@fakeemail.com
Street: 1 Main St.
City/State/Zip: Some Town, MA 02215

Touching a contact's name launches an `Activity` that displays the contact's details

Fig. 10.1 | List of contacts with one item touched and the detailed contact information for the touched contact.

taining an **Add Contact** option—touching that option launches an `Activity` for adding a new contact (Fig. 10.4). Touching the **Save Contact** `Button` adds the new contact and returns the user to the main contact screen.

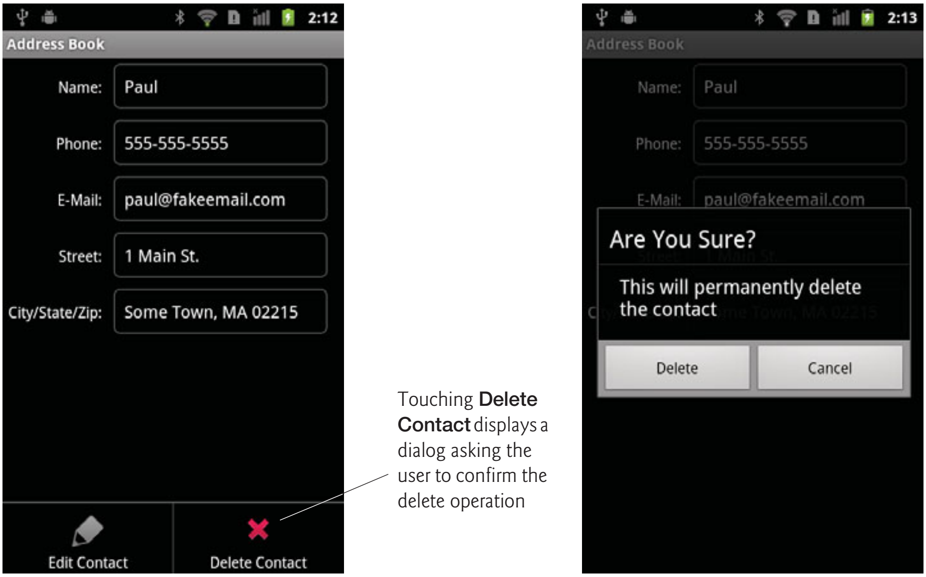


Name: Paul
Phone: 555-555-5555
E-Mail: paul@fakeemail.com
Street: 1 Main St.
City/State/Zip: Some Town, MA 02215

Paul
555-555-5555
paul@fakeemail.com
Address
1 Main St.
Some Town, MA 02215

Touching **Edit Contact** launches an `Activity` for editing that contact's data

Fig. 10.2 | Editing a contact's data.

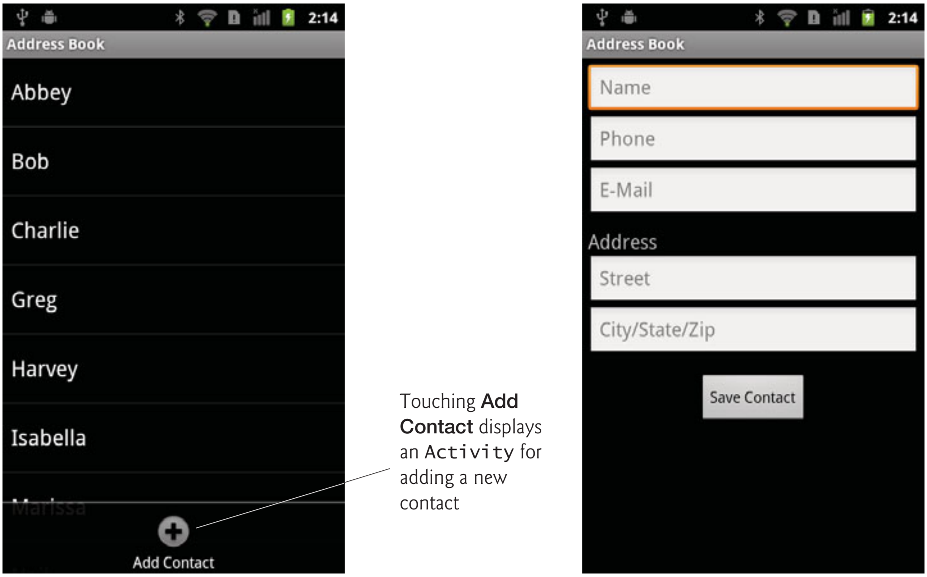


Name: Paul
Phone: 555-555-5555
E-Mail: paul@fakeemail.com
Street: 1 Main St.
City/State/Zip: Some Town, MA 02215

Are You Sure?
This will permanently delete the contact

Touching **Delete Contact** displays a dialog asking the user to confirm the delete operation

Fig. 10.3 | Deleting a contact from the database.



Touching **Add Contact** displays an `Activity` for adding a new contact

Fig. 10.4 | Adding a contact to the database.

10.2 Test-Driving the Address Book App

Opening and Running the App

Open Eclipse and import the **Address Book** app project. To import the project:

1. Select **File > Import…** to display the **Import** dialog.
2. Expand the **General** node and select **Existing Projects into Workspace**, then click **Next >**.
3. To the right of the **Select root directory:** text field, click **Browse…**, then locate and select the `AddressBook` folder.
4. Click **Finish** to import the project.

Right click the app's project in the **Package Explorer** window, then select **Run As > Android Application** from the menu that appears.

Adding a Contact

The first time you run the app, the contact list will be empty. Touch the device's menu button, then touch **Add Contact** to display the screen for adding a new entry. After adding the contact's information, touch the **Save Contact** `Button` to store the contact in the database and return to the app's main screen. If you choose not to add the contact, you can simply touch the device's back button to return to the main screen. Add more contacts if you wish.

Viewing a Contact

Touch the name of the contact you just added in the contacts list to view that contact's details.

Editing a Contact

While viewing the contact's details, touch the device's menu button then touch **Edit Contact** to display a screen of `EditText`s that are prepopulated with the contact's data. Edit the data as necessary then touch the **Save Contact** `Button` to store the updated contact information in the database and return to the app's main screen.

Deleting a Contact

While viewing the contact's details, touch the device's menu button, then touch **Delete Contact**. If you wish to delete the contact, confirm this action in the dialog. The contact will be removed from the database and the app will return to the main screen.

Android 2.3 Overscroll

As of Android 2.3, lists like the one used to display the contacts in this app support **overscroll**—a visual effect (orange highlight) that indicates when you've reached the top or bottom of the list while scrolling through its contents. You can see the orange highlight effect by attempting to scroll past the beginning or end of the list.

10.3 Technologies Overview

This section presents the new technologies that we use in the **Address Book** app in the order in which they're encountered throughout the chapter.

Specifying Additional `activity` Elements in the App's Manifest

The `AndroidManifest.xml` file describes an app's components. In the prior apps, we had only one `Activity` per app. In this app, we have three. Each `Activity` must be described in the app's manifest (Section 10.4.2).

Defining Styles and Applying Them to GUI Components

You can define common GUI component attribute–value pairs as XML **`style` resources** (Section 10.4.3). You can then apply the styles to all components that share those values (Section 10.4.6) by using the **`style` attribute**. Any subsequent changes you make to a `style` are automatically applied to all GUI components that use the `style`.

Specifying a Background for a `TextView`

By default `TextView`s do not have a border. To define one, you can specify a `Drawable` as the value for the `TextView`'s `android:background` attribute. The `Drawable` could be an image, but in this app we'll define a new type of `Drawable` using an XML representation of a `shape` (Section 10.4.4). The XML file for such a `Drawable` is placed in the app's `drawable` folder, which you must create in the app's `res` folder.

Specifying the Format of a `ListView`'s Items

This app uses a **`ListView`** (package `android.widget`) to display the contact list as a list of items that is *scrollable* if the complete list cannot be displayed on the screen. You can specify the layout resource (Section 10.4.5) that will be used to display each `ListView` item.

Creating `menu` Resources in XML and Inflating Them with a `MenuInflater`

In previous apps that used menus, we programmatically created the `MenuItem`s. In this app, we'll use **`menu` resources** in XML to define the `MenuItem`s, then we'll programmatically inflate them (Sections 10.5.1 and 10.5.2) using an Activity's **MenuInflater** (package android.view), which is similar to a LayoutInflater. In addition, we'll use some of Android's standard icons to enhance the visual appearance of the menu items.

*Extending Class **ListActivity** to Create an **Activity** That Contains a **ListView***

When an Activity's primary task is to display a scrollable list of items, you can extend class **ListActivity** (package android.app, Section 10.5.1), which uses a ListView that occupies the entire screen as its default layout. ListView is a subclass of **AdapterView** (package android.widget)—a GUI component is bound to a data source via an **Adapter** object (package android.widget). In this app, we'll use a **CursorAdapter** (package android.widget) to display the results of a database query in the ListView.

Several types of AdapterViews can be bound to data using an Adapter. For more details on data binding in Android and several tutorials, visit

```
    developer.android.com/guide/topics/ui/binding.html
```

*Using an Explicit **Intent** to Launch Another **Activity** in the Same App and Passing Data to That **Activity***

This app allows the user to view an existing contact, add a new contact or edit an existing contact. In each case, we launch a new Activity to handle the specified task. In Chapter 5, we showed how to use an *implicit* Intent to display a URL in the device's web browser. Sections 10.5.1 and 10.5.2 show how to use **explicit Intents** to launch another Activity in the same app and how to pass data from one Activity to another. Section 10.5.3 shows how to return to the Activity that launched a particular Activity.

Manipulating a SQLite Database

This app's contact information is stored in a SQLite database. SQLite (www.sqlite.org) is the world's most widely deployed database engine. Each Activity in this app interacts with the SQLite database via our utility class DatabaseConnector (Section 10.5.4). Within that class, we use a nested subclass of **SQLiteOpenHelper** (package **android.database.sqlite**), which simplifies creating the database and enables you to obtain a **SQLiteDatabase** object (package android.database.sqlite) for manipulating a database's contents. Database query results are managed via a **Cursor** (package **android.database**).

Using Multithreading to Perform Database Operations Outside the GUI Thread

It's good practice to perform long running operations or operations that block execution until they complete (e.g., file and database access) outside the GUI thread. This helps maintain application responsiveness and avoid *Activity Not Responding (ANR) dialogs* that appear when Android thinks the GUI is not responsive. When we need a database operation's results in the GUI thread, we'll use an **AsyncTask** (package android.os) to perform the operation in one thread and receive the results in the GUI thread. The details of creating and manipulating threads are handled for you by class AsyncTask, as are communicating the results from the AsyncTask to the GUI thread.

10.4 Building the GUI and Resource Files

In this section, you'll create the **Address Book** app's resource files and GUI layout files. To save space, we do not show this app's strings.xml resource file or the layout files for the

ViewContact Activity (view_contact.xml) and AddEditContact (add_contact.xml). You can view the contents of these files by opening them from the project in Eclipse.

10.4.1 Creating the Project

Begin by creating a new Android project named AddressBook. Specify the following values in the **New Android Project** dialog, then press **Finish**:

- **Build Target**: Ensure that **Android 2.3.3** is checked
- **Application name**: Address Book
- **Package name**: com.deitel.addressbook
- **Create Activity**: AddressBook
- **Min SDK Version**: 8

10.4.2 AndroidManifest.xml

Figure 10.5 shows this app's AndroidManifest.xml file, which contains an activity element for each Activity in the app. Lines 14–15 specify AddEditContact's activity element. Lines 16–17 specify ViewContact's activity element.

```
 1  <?xml version="1.0" encoding="utf-8"?>
 2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
 3     package="com.deitel.addressbook" android:versionCode="1"
 4     android:versionName="1.0">
 5     <application android:icon="@drawable/icon"
 6        android:label="@string/app_name">
 7        <activity android:name=".AddressBook"
 8           android:label="@string/app_name">
 9           <intent-filter>
10              <action android:name="android.intent.action.MAIN" />
11              <category android:name="android.intent.category.LAUNCHER" />
12           </intent-filter>
13        </activity>
14        <activity android:name=".AddEditContact"
15           android:label="@string/app_name"></activity>
16        <activity android:name=".ViewContact"
17           android:label="@string/app_name"></activity>
18     </application>
19     <uses-sdk android:minSdkVersion="8" />
20  </manifest>
```

Fig. 10.5 | AndroidManifest.xml.

10.4.3 styles.xml

Figure 10.6 defines the style resources used in the layout file view_contact.xml (Section 10.4.6). Like XML documents representing other values, an XML document containing style elements is placed in the app's res/values folder. Each style specifies a name (e.g., line 3), which is used to apply that style to one or more GUI components, and to one or more item elements (e.g., line 4), each specifying an attribute's XML name and a value to apply.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <resources>
3     <style name="ContactLabelTextView">
4        <item name="android:layout_width">wrap_content</item>
5        <item name="android:layout_height">wrap_content</item>
6        <item name="android:gravity">right</item>
7        <item name="android:textSize">14sp</item>
8        <item name="android:textColor">@android:color/white</item>
9        <item name="android:layout_marginLeft">5dp</item>
10       <item name="android:layout_marginRight">5dp</item>
11       <item name="android:layout_marginTop">5dp</item>
12    </style>
13    <style name="ContactTextView">
14       <item name="android:layout_width">wrap_content</item>
15       <item name="android:layout_height">wrap_content</item>
16       <item name="android:textSize">16sp</item>
17       <item name="android:textColor">@android:color/white</item>
18       <item name="android:layout_margin">5dp</item>
19       <item name="android:background">@drawable/textview_border</item>
20    </style>
21 </resources>
```

Fig. 10.6 | Styles defined in `styles.xml` and placed in the app's `res/values` folder.

10.4.4 `textview_border.xml`

The `style` `ContactTextView` in Fig. 10.6 (lines 13–20) defines the appearance of the `TextView`s that are used to display a contact's details in the `ViewContact` `Activity`. Line 19 specifies a `Drawable` as the value for the `TextView`'s `android:background` attribute. The `Drawable` (`textview_border`) used here is defined in XML as a **`shape` element** (Fig. 10.7) and stored in the app's `res/drawable` folder. The `shape` element's `android:shape` attribute (line 3) can have the value `"rectangle"` (used in this example), `"oval"`, `"line"` or `"ring"`. The **`corners` element** (line 4) specifies the rectangle's corner radius, which rounds the corners. The **`stroke` element** (line 5) defines the rectangle's line width and line color. The **`padding` element** (lines 6–7) specifies the spacing around the content in the element to which this `Drawable` is applied. You must specify the top, left, right and bottom padding amounts separately. The complete specification for defining a shape in XML can be viewed at:

```
developer.android.com/guide/topics/resources/
   drawable-resource.html#Shape
```

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <shape xmlns:android="http://schemas.android.com/apk/res/android"
3     android:shape="rectangle" >
4     <corners android:radius="5dp"/>
5     <stroke android:width="1dp" android:color="#555"/>
6     <padding android:top="10dp" android:left="10dp" android:bottom="10dp"
7        android:right="10dp"/>
8  </shape>
```

Fig. 10.7 | XML representation of a `Drawable` that's used to place a border on a `TextView`.

10.4.5 AddressBook Activity's Layout: contact_list_item.xml

The AddressBook Activity extends ListActivity rather than Activity. A ListActivity's default GUI consists of a ListView that occupies the entire screen, so we do not need to define a separate layout for this Activity. If you wish to customize a ListActivity's GUI, you can define a layout XML file that must contain a ListView with its android:id attribute set to "@android:id/list", which we discuss in Chapter 12's **Slideshow** app.

When populating a ListView with data, you must specify the format that's applied to each list item, which is the purpose of the contact_list_item.xml layout in Fig. 10.8. Each list item contains one contact's name, so the layout defines just a TextView for displaying a name. A ListView's default background color is black, so we set the text color to white (line 5). The android:id attribute will be used to associate data with the TextView. Line 6 sets the list item's minimum height to listPreferredItemHeight—a built in Android attribute constant. Line 7 sets the list item's gravity to center_vertical. If a list item should consist of multiple pieces of data, you may need multiple elements in your list-item layout and each will need an android:id attribute. You'll learn how to use these android:id attributes in Section 10.5.1. Figure 10.1 showed the list-items' appearance.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <TextView xmlns:android="http://schemas.android.com/apk/res/android"
3     android:id="@+id/contactTextView" android:layout_width="match_parent"
4     android:layout_height="wrap_content" android:padding="8dp"
5     android:textSize="20sp" android:textColor="@android:color/white">
6     android:minHeight="?android:attr/listPreferredItemHeight"
7     android:gravity="center_vertical"></TextView>
```

Fig. 10.8 | Layout for each item in the AddressBook ListActivity's built-in ListView.

10.4.6 ViewContact Activity's Layout: view_contact.xml

When the user selects a contact in the AddressBook Activity, the app launches the ViewContact Activity (Fig. 10.9). This Activity's layout (view_contact.xml) uses a ScrollView containing a TableLayout in which each TableRow contains two TextViews.

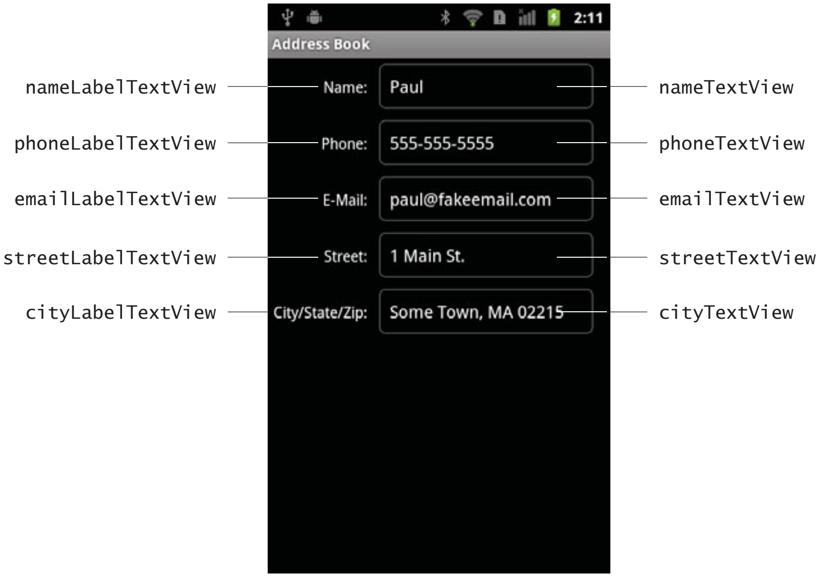
The only new feature in this layout is that all of its TextViews have styles from Fig. 10.6 applied to them. For example, lines 11–15 in the layout file:

```
<TextView android:id="@+id/nameLabelTextView"
   style="@style/ContactLabelTextView"
   android:text="@string/label_name"></TextView>
<TextView android:id="@+id/nameTextView"
   style="@style/ContactTextView"></TextView>
```

represent the TextViews in the first TableRow. Each TextView uses the style attribute to specify the style to apply using the syntax @style/*styleName*.

10.4.7 AddEditContact Activity's Layout: add_contact.xml

When the user touches the AddressBook Activity's **Add Contact** menu item or the ViewContact Activity's **Edit Contact** menu item, the app launches the AddEditContact Activity (Fig. 10.10). This Activity's layout uses a ScrollView containing a vertical LinearLayout. If the Activity is launched from the AddressBook Activity, the Edit-



2:11

Address Book

Name: Paul

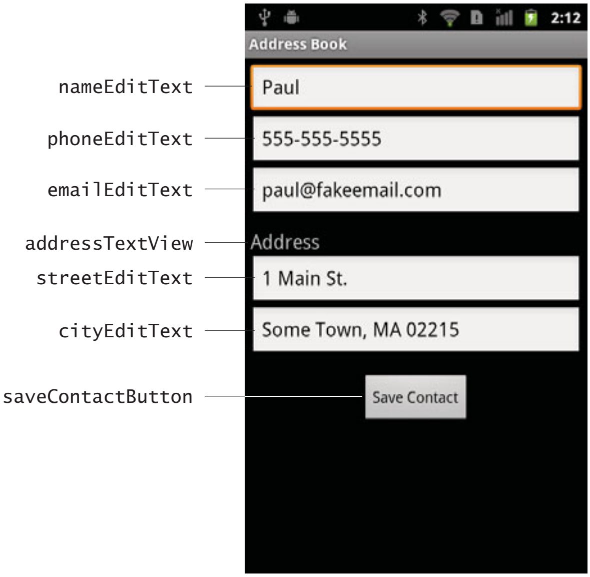
Phone: 555-555-5555

E-Mail: paul@fakeemail.com

Street: 1 Main St.

City/State/Zip: Some Town, MA 02215

Fig. 10.9 | `ViewContact` `Activity`'s GUI components labeled with their `id` property values. This GUI's root component is a `ScrollView` containing a `TableLayout` with five `TableRows`.



2:12

Address Book

Paul

555-555-5555

paul@fakeemail.com

Address

1 Main St.

Some Town, MA 02215

Fig. 10.10 | `AddEditContact` `Activity`'s GUI components labeled with their `id` property values. This GUI's root component is a `ScrollView` that contains a vertical `LinearLayout`.

`Texts` will be empty and will display hints (specified in lines 12, 17, 22, 33 and 38 of the layout's XML file). Otherwise, the `EditTexts` will display the contact's data that was passed to the AddEditContact Activity from the ViewContact Activity. Each EditText specifies the android:inputType and android:imeOptions attributes. For devices that display a soft keyboard, the android:inputType attribute (at lines 13, 18, 23, 34 and 39 in the layout's XML file) specifies which keyboard to display when the user touches the corresponding EditText. This enables us to *customize the keyboard* to the specific type of data the user must enter in a given EditText. As in Chapter 5, we use the android:imeOptions attribute to display a **Next** button on the soft keyboards for the nameEditText, emailEditText, phoneEditText or streetEditText. When one of these has the focus, touching this Button transfers the focus to the next EditText. If the cityEditText has the focus, you can hide the soft keyboard by touching the keyboard's **Done** Button.

10.4.8 Defining the App's MenuItems with menu Resources in XML

Figures 10.11 and 10.12 define the menu resources for the AddressBook Activity and the ViewContact Activity, respectively. Resource files that define menus are placed in the app's res/menu folder (which you must create) and are added to the project like other resource files (originally described in Section 3.5), but in the **New Android XML File** dialog you select **Menu** as the resource type. Each menu resource XML file contains a root **menu element** with nested **item elements** that represent each MenuItem. We show how to inflate the menus in Sections 10.5.1 and 10.5.2.

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

Fig. 10.11 | AddressBook Activity's menu resource.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
   <item android:id="@+id/editItem"
      android:title="@string/menuitem_edit_contact"
      android:orderInCategory="1" android:alphabeticShortcut="e"
      android:titleCondensed="@string/menuitem_edit_contact"
      android:icon="@android:drawable/ic_menu_edit"></item>
   <item android:id="@+id/deleteItem"
      android:title="@string/menuitem_delete_contact"
      android:orderInCategory="2" android:alphabeticShortcut="d"
      android:titleCondensed="@string/menuitem_delete_contact"
      android:icon="@android:drawable/ic_delete"></item>
</menu>
```

Fig. 10.12 | ViewContact Activity's menu resource.

You specify an android:id attribute for each item so that you can interact with the corresponding MenuItem programmatically. Other item attributes we use here include:

- **android:title** and **android:titleCondensed**—these specify the text to display on the MenuItem. The condensed title is used if the regular title text is too long to display properly.
- **android:icon**—specifies a Drawable to display on the MenuItem above the title text. In this example's MenuItems, we use three of the standard icons that are provided with the Android SDK. They're located in the SDK's platforms folder under each platform version's data/res/drawable-hdpi folder. To refer to these icons in your XML layouts, prefix them with @android:drawable/*icon_name* as in Fig. 10.11, line 5 and Fig. 10.12, lines 7 and 12.
- **android:alphabeticShortcut**—specifies a letter that the user can press on a hard keyboard to select the menu item.
- **android:orderInCategory**—determines the order in which the MenuItems appear. We did not use it in Fig. 10.11, as there's only one MenuItem.

For complete details on menu resources, visit:

```
developer.android.com/guide/topics/resources/menu-resource.html
```

10.5 Building the App

This app consists of four classes—class AddressBook (the ListActivity subclass, Figs. 10.13–10.18), class ViewContact (Figs. 10.19–10.23), class AddEditContact (Figs. 10.24–10.27) and class DatabaseConnector (Figs. 10.28–10.31). As in prior apps, this app's main Activity—AddressBook—is created when you create the project, but you'll need to modify it to extend class ListActivity. You must add the other Activity classes and the DatabaseConnector class to the project's src/com.deitel.addressbook folder.

10.5.1 AddressBook Subclass of ListActivity

Class AddressBook (Figs. 10.13–10.18) provides the functionality for the first Activity displayed by this app. As discussed earlier in this chapter, the class extends ListActivity rather than Activity, because this Activity's primary purpose is to display a ListView containing the user's contacts.

package *Statement,* **import** *Statements and Instance Variables*

Figure 10.13 lists AddressBook's package statement, import statements and instance variables. We've highlighted the imports for the new classes discussed in Section 10.3. The constant ROW_ID is used as a key in a key–value pair that's passed between activities (Fig. 10.18). Instance variable contactListView will refer to the AddressBook's built-in ListView, so we can interact with it programmatically. Instance variable contactAdapter will refer to the CursorAdapter that populates the AddressBook's ListView.

```
// AddressBook.java
// Main activity for the Address Book app.
package com.deitel.addressbook;
```

Fig. 10.13 | package statement, import statements and instance variables of class AddressBook. (Part 1 of 2.)

```
 4
 5  import android.app.ListActivity;
 6  import android.content.Intent;
 7  import android.database.Cursor;
 8  import android.os.AsyncTask;
 9  import android.os.Bundle;
10  import android.view.Menu;
11  import android.view.MenuInflater;
12  import android.view.MenuItem;
13  import android.view.View;
14  import android.widget.AdapterView;
15  import android.widget.AdapterView.OnItemClickListener;
16  import android.widget.CursorAdapter;
17  import android.widget.ListView;
18  import android.widget.SimpleCursorAdapter;
19
20  public class AddressBook extends ListActivity
21  {
22     public static final String ROW_ID = "row_id"; // Intent extra key
23     private ListView contactListView; // the ListActivity's ListView
24     private CursorAdapter contactAdapter; // adapter for ListView
25
```

Fig. 10.13 | package statement, import statements and instance variables of class AddressBook. (Part 2 of 2.)

Overriding Activity *Method* onCreate

Method onCreate (Fig. 10.14, lines 26–32) initializes the Activity. Recall that class ListActivity already contains a ListView that occupies the entire Activity, we don't need to inflate the GUI using method setContentView as in previous apps. Line 31 uses the inherited ListActivity method **getListView** to obtain a reference to the built-in ListView. Line 32 then sets the ListView's OnItemClickListener to viewContactListener (Fig. 10.18), which responds to the user's touching one of the ListView's items.

```
26     // called when the activity is first created
27     @Override
28     public void onCreate(Bundle savedInstanceState)
29     {
30        super.onCreate(savedInstanceState); // call super's onCreate
31        contactListView = getListView(); // get the built-in ListView
32        contactListView.setOnItemClickListener(viewContactListener);
33
34        // map each contact's name to a TextView in the ListView layout
35        String[] from = new String[] { "name" };
36        int[] to = new int[] { R.id.contactTextView };
37        CursorAdapter contactAdapter = new SimpleCursorAdapter(
38           AddressBook.this, R.layout.contact_list_item, null, from, to);
39        setListAdapter(contactAdapter); // set contactView's adapter
40     } // end method onCreate
41
```

Fig. 10.14 | Overriding Activity method onCreate.

To display the Cursor's results in a ListView we create a new CursorAdapter object (lines 35–38) which exposes the Cursor's data in a manner that can be used by a ListView. **SimpleCursorAdapter** is a subclass of CursorAdapter that's designed to simplify mapping Cursor columns directly to TextViews or ImagesViews defined in your XML layouts. To create a SimpleCursorAdapter, you must first define arrays containing the column names to map to GUI components and the resource IDs of the GUI components that will display the data from the named columns. Line 35 creates a String array indicating that only the column named name will be displayed, and line 36 creates a parallel int array containing corresponding GUI components' resource IDs (in this case, R.id.contactTextView). Lines 37–38 create the SimpleCursorAdapter. Its constructor receives:

- the Context in which the ListView is running (i.e., the AddressBook Activity)
- the resource ID of the layout that's used to display each item in the ListView
- the Cursor that provides access to the data—we supply null for this argument because we'll specify the Cursor later
- the String array containing the column names to display
- the int array containing the corresponding GUI resource IDs

Line 39 uses inherited ListActivity method **setListAdapter** to bind the ListView to the CursorAdapter, so that the ListView can display the data.

Overriding **Activity** *Methods* **onResume** *and* **onStop**

As you learned in Section 8.5.1, method onResume (Fig. 10.15, lines 42–49) is called each time an Activity returns to the foreground, including when the Activity is first created. In this app, onResume creates and executes an AsyncTask (line 48) of type GetContactsTask (defined in Fig. 10.16) that gets the complete list of contacts from the database and sets the contactAdapter's Cursor for populating the AddressBook's ListView. AsyncTask method **execute** performs the task in a separate thread. Method execute's argument in this case indicates that the task does not receive any arguments— this method can receive a variable number of arguments that are, in turn, passed as arguments to the task's doInBackground method. Every time line 48 executes, it creates a new GetContactsTask object—this is required because each AsyncTask can be executed *only once*.

```
42    @Override
43    protected void onResume()
44    {
45       super.onResume(); // call super's onResume method
46
47       // create new GetContactsTask and execute it
48       new GetContactsTask().execute((Object[]) null);
49    } // end method onResume
50
51    @Override
52    protected void onStop()
53    {
54       Cursor cursor = contactAdapter.getCursor(); // get current Cursor
```

Fig. 10.15 | Overriding Activity methods onResume and onStop. (Part 1 of 2.)

```
55
56         if (cursor != null)
57            cursor.deactivate(); // deactivate it
58
59         contactAdapter.changeCursor(null); // adapted now has no Cursor
60         super.onStop();
61      } // end method onStop
62
```

Fig. 10.15 | Overriding `Activity` methods `onResume` and `onStop`. (Part 2 of 2.)

`Activity` method **`onStop`** (Fig. 10.15, lines 51–61) is called when the `Activity` is no longer visible to the user—typically because another `Activity` has started or returned to the foreground. In this case, the `Cursor` that allows us to populate the `ListView` is not needed, so line 54 calls `CursorAdapter` method **`getCursor`** to get the current `Cursor` from the `contactAdapter`, then line 57 calls `Cursor` method **`deactivate`** to release resources used by the `Cursor`. Line 59 then calls `CursorAdapter` method **`changeCursor`** with the argument `null` to remove the `Cursor` from the `CursorAdapter`.

`GetContactsTask` Subclass of `AsyncTask`

Nested class `GetContactsTask` (Fig. 10.16) extends class `AsyncTask`. The class defines how to interact with the database to get the names of all the contacts and return the results to this `Activity`'s GUI thread for display in the `ListView`. `AsyncTask` is a generic type that requires three type parameters:

- The first is the type of the variable length parameter list for the `AsyncTask`'s **`doInBackground`** method (lines 50–57). When an `AsyncTask`'s `execute` method is called, the task's `doInBackground` method performs the task in a separate thread of execution. In this case, `doInBackground` does not require additional data to perform its task, so we specify `Object` as the type parameter and pass `null` as the argument to the `AsyncTask`'s `execute` method, which calls `doInBackground`.
- The second is the type of the variable length parameter list for the `AsyncTask`'s **`onProgressUpdate`** method. This method executes in the GUI thread and is used to receive intermediate updates of the specified type from a long-running task. We don't use this feature in this example, so we specify type `Object` here and ignore this type parameter.
- The third is the type of the task's result, which is passed to the `AsyncTask`'s **`onPostExecute`** method (lines 80–85). This method executes in the GUI thread and enables the `Activity` to use the `AsyncTask`'s results.

A key benefit of using an `AsyncTask` is that it handles the details of creating threads and executing its methods on the appropriate threads for you, so that you do not have to interact with the threading mechanism directly.

Lines 66–67 create a new object of our utility class `DatabaseConnector`, passing the `Context` (`AddressBook.this`) as an argument to the class's constructor. (We discuss class `DatabaseConnector` in Section 10.5.4.)

Method `doInBackground` (lines 70–77) uses `databaseConnector` to open the database connection, then gets all the contacts from the database. The `Cursor` returned by

```
63      // performs database query outside GUI thread
64      private class GetContactsTask extends AsyncTask<Object, Object, Cursor>
65      {
66         DatabaseConnector databaseConnector =
67            new DatabaseConnector(AddressBook.this);
68
69         // perform the database access
70         @Override
71         protected Cursor doInBackground(Object... params)
72         {
73            databaseConnector.open();
74
75            // get a cursor containing call contacts
76            return databaseConnector.getAllContacts();
77         } // end method doInBackground
78
79         // use the Cursor returned from the doInBackground method
80         @Override
81         protected void onPostExecute(Cursor result)
82         {
83            contactAdapter.changeCursor(result); // set the adapter's Cursor
84            databaseConnector.close();
85         } // end method onPostExecute
86      } // end class GetContactsTask
87
```

Fig. 10.16 | GetContactsTask subclass of AsyncTask

getAllContacts is passed to method onPostExecute (lines 80–86). That method receives the Cursor containing the results, and passes it to CursorAdapter method changeCursor, so the Activity's ListView can populate itself.

Managing Cursors

In this Activity, we're managing the Cursors with various Cursor and CursorAdapter methods. Class Activity can also manage Cursors for you. Activity method **startManagingCursor** tells the Activity to manage the Cursor's lifecycle based on the Activity's lifecycle. When the Activity is stopped, it will call deactivate on any Cursors it's currently managing. When the Activity resumes, it will call **requery** on its Cursors. When the Activity is destroyed, it will automatically call close to *release all resources* held by any managed Cursors. A deactivated Cursor consumes less resources than an active one, so it's good practice to align your Cursor's lifecycle with its parent Activity if the Cursor is not shared among multiple Activity objects. Allowing your Activity to manage the Cursor's lifecycle also ensures that the Cursor will be closed when it's no longer needed.

Overriding Activity *Methods* onCreateOptionsMenu *and* onOptionsItemSelected

When the user opens this Activity's menu, method onCreateOptionsMenu (Fig. 10.17, lines 89–96) uses a **MenuInflater** to create the menu from addressbook_menu.xml, which contains an **Add Contact** MenuItem. We obtain the MenuInflater by calling Activity's **getMenuInflater method**. If the user touches that MenuItem, method onOptionsItemSelected (lines 99–107) launches the AddEditContact Activity (Section 10.5.3). Lines

103–104 create a new *explicit* Intent to launch that Activity. The Intent constructor used here receives the Context from which the Activity will be launched and the class representing the Activity to launch (AddEditContact.class). We then pass this Intent to the inherited Activity method startActivity to launch the Activity.

```
88     // create the Activity's menu from a menu resource XML file
89     @Override
90     public boolean onCreateOptionsMenu(Menu menu)
91     {
92        super.onCreateOptionsMenu(menu);
93        MenuInflater inflater = getMenuInflater();
94        inflater.inflate(R.menu.addressbook_menu, menu);
95        return true;
96     } // end method onCreateOptionsMenu
97
98     // handle choice from options menu
99     @Override
100    public boolean onOptionsItemSelected(MenuItem item)
101    {
102       // create a new Intent to launch the AddEditContact Activity
103       Intent addNewContact =
104          new Intent(AddressBook.this, AddEditContact.class);
105       startActivity(addNewContact); // start the AddEditContact Activity
106       return super.onOptionsItemSelected(item); // call super's method
107    } // end method onOptionsItemSelected
108
```

Fig. 10.17 | Overriding Activity methods onCreateOptionsMenu and onOptionsItemSelected.

Anonymous Inner Class That Implements Interface ***OnItemClickListener*** *to Process* ***ListView*** *Events*

The viewContactListener OnItemClickListener (Fig. 10.18) launches the ViewContact Activity to display the user's selected contact. Method **onItemClick** receives:

- a reference to the AdapterView that the user interacted with (i.e., the ListView),
- a reference to the root View of the touched list item,
- the index of the touched list item in the ListView and
- the unique long ID of the selected item—in this case, the row ID in the Cursor.

```
   // event listener that responds to the user touching a contact's name
   // in the ListView
   OnItemClickListener viewContactListener = new OnItemClickListener()
   {
      @Override
      public void onItemClick(AdapterView<?> arg0, View arg1, int arg2,
         long arg3)
      {
```

Fig. 10.18 | OnItemClickListener viewContactListener that responds to ListView touch events. (Part 1 of 2.)

```
117          // create an Intent to launch the ViewContact Activity
118          Intent viewContact =
119             new Intent(AddressBook.this, ViewContact.class);
120
121          // pass the selected contact's row ID as an extra with the Intent
122          viewContact.putExtra(ROW_ID, arg3);
123          startActivity(viewContact); // start the ViewContact Activity
124       } // end method onItemClick
125    }; // end viewContactListener
126 } // end class AddressBook
```

Fig. 10.18 | `OnItemClickListener viewContactListener` that responds to `ListView` touch events. (Part 2 of 2.)

Lines 118–119 create an explicit `Intent` to launch the `ViewContact Activity`. To display the appropriate contact, the `ViewContact Activity` needs to know which record to retrieve. You can pass data between activities by adding *extras* to the `Intent` using `Intent`'s **`putExtra`** method (line 122), which adds the data as a key–value pair to a `Bundle` associated with the `Intent`. In this case, the key–value pair represents the unique row ID of the contact the user touched.

10.5.2 `ViewContact` Subclass of `Activity`

The `ViewContact Activity` (Figs. 10.19–10.23) displays one contact's information and provides a menu that enables the user to edit or delete that contact.

`package`* *Statement,* *`import`* *Statements and Instance Variables

Figure 10.19 lists the `package` statement, the `import` statements and the instance variables for class `ViewContact`. We've highlighted the `import` statements for the new classes discussed in Section 10.3. The instance variable `rowID` represents the current contact's unique row ID in the database. The `TextView` instance variables (lines 20–24) are used to display the contact's data on the screen.

```
1  // ViewContact.java
2  // Activity for viewing a single contact.
3  package com.deitel.addressbook;
4
5  import android.app.Activity;
6  import android.app.AlertDialog;
7  import android.content.DialogInterface;
8  import android.content.Intent;
9  import android.database.Cursor;
10 import android.os.AsyncTask;
11 import android.os.Bundle;
12 import android.view.Menu;
13 import android.view.MenuInflater;
14 import android.view.MenuItem;
```

Fig. 10.19 | `package` statement, `import` statements and instance variables of class `ViewContact`. (Part 1 of 2.)

```
15 import android.widget.TextView;
16
17 public class ViewContact extends Activity
18 {
19    private long rowID; // selected contact's name
20    private TextView nameTextView; // displays contact's name
21    private TextView phoneTextView; // displays contact's phone
22    private TextView emailTextView; // displays contact's email
23    private TextView streetTextView; // displays contact's street
24    private TextView cityTextView; // displays contact's city/state/zip
25
```

Fig. 10.19 | package statement, import statements and instance variables of class ViewContact. (Part 2 of 2.)

Overriding Activity *Methods* onCreate *and* onResume

The onCreate method (Fig. 10.20, lines 27–43) first gets references to the Activity's TextViews, then obtains the selected contact's row ID. Activity method **getIntent** returns the Intent that launched the Activity. We use that to call Intent method **getExtras**, which returns a Bundle that contains any key–value pairs that were added to the Intent as extras. This method returns null if no extras were added. Next, we use the Bundle's **getLong** method to obtain the long integer representing the selected contact's row ID. [*Note:* We did not test whether the value of extras (line 41) was null, because there will always be a Bundle returned in this app. Testing for null is considered good practice, so you can decide how to handle the problem. For example, you could log the error and return from the Activity by calling finish.] Method onResume (lines 46–53) simply creates a new AsyncTask of type LoadContactTask (Fig. 10.21) and executes it to get and display contact's information.

```
26    // called when the activity is first created
27    @Override
28    public void onCreate(Bundle savedInstanceState)
29    {
30       super.onCreate(savedInstanceState);
31       setContentView(R.layout.view_contact);
32
33       // get the EditTexts
34       nameTextView = (TextView) findViewById(R.id.nameTextView);
35       phoneTextView = (TextView) findViewById(R.id.phoneTextView);
36       emailTextView = (TextView) findViewById(R.id.emailTextView);
37       streetTextView = (TextView) findViewById(R.id.streetTextView);
38       cityTextView = (TextView) findViewById(R.id.cityTextView);
39
40       // get the selected contact's row ID
41       Bundle extras = getIntent().getExtras();
42       rowID = extras.getLong("row_id");
43    } // end method onCreate
44
```

Fig. 10.20 | Overriding Activity method onCreate. (Part 1 of 2.)

```
   // called when the activity is first created
   @Override
   protected void onResume()
   {
      super.onResume();

      // create new LoadContactTask and execute it
      new LoadContactTask().execute(rowID);
   } // end method onResume

```

Fig. 10.20 | Overriding `Activity` method `onCreate`. (Part 2 of 2.)

`GetContactsTask` *Subclass of* `AsyncTask`

Nested class `GetContactsTask` (Fig. 10.21) extends class `AsyncTask` and defines how to interact with the database and get one contact's information for display. In this case the three generic type parameters are:

- `Long` for the variable-length argument list passed to `AsyncTask`'s `doInBackground` method. This will contain the row ID needed to locate one contact.
- `Object` for the variable-length argument list passed to `AsyncTask`'s `onProgressUpdate` method, which we don't use in this example.
- `Cursor` for the type of the task's result, which is passed to the `AsyncTask`'s `onPostExecute` method.

```
   // performs database query outside GUI thread
   private class LoadContactTask extends AsyncTask<Long, Object, Cursor>
   {
      DatabaseConnector databaseConnector =
         new DatabaseConnector(ViewContact.this);

      // perform the database access
      @Override
      protected Cursor doInBackground(Long... params)
      {
         databaseConnector.open();

         // get a cursor containing all data on given entry
         return databaseConnector.getOneContact(params[0]);
      } // end method doInBackground

      // use the Cursor returned from the doInBackground method
      @Override
      protected void onPostExecute(Cursor result)
      {
         super.onPostExecute(result);

         result.moveToFirst(); // move to the first item

```

Fig. 10.21 | `loadContact` method of class `ViewContact`. (Part 1 of 2.)

```
79          // get the column index for each data item
80          int nameIndex = result.getColumnIndex("name");
81          int phoneIndex = result.getColumnIndex("phone");
82          int emailIndex = result.getColumnIndex("email");
83          int streetIndex = result.getColumnIndex("street");
84          int cityIndex = result.getColumnIndex("city");
85
86          // fill TextViews with the retrieved data
87          nameTextView.setText(result.getString(nameIndex));
88          phoneTextView.setText(result.getString(phoneIndex));
89          emailTextView.setText(result.getString(emailIndex));
90          streetTextView.setText(result.getString(streetIndex));
91          cityTextView.setText(result.getString(cityIndex));
92
93          result.close(); // close the result cursor
94          databaseConnector.close(); // close database connection
95       } // end method onPostExecute
96    } // end class LoadContactTask
97
```

Fig. 10.21 | `loadContact` method of class `ViewContact`. (Part 2 of 2.)

Lines 58–59 create a new object of our `DatabaseConnector` class (Section 10.5.4). Method `doInBackground` (lines 62–69) opens the connection to the database and calls the `DatabaseConnector`'s `getOneContact` method, which queries the database to get the contact with the specified `rowID` that was passed as the only argument to this `AsyncTask`'s `execute` method. In `doInBackground`, the `rowID` is stored in `params[0]`.

The resulting `Cursor` is passed to method `onPostExecute` (lines 72–95). The `Cursor` is positioned *before* the first row of the result set. In this case, the result set will contain only one record, so `Cursor` method **`moveToFirst`** (line 77) can be used to move the `Cursor` to the first row in the result set. [*Note:* It's considered good practice to ensure that `Cursor` method `moveToFirst` returns `true` before attempting to get data from the `Cursor`. In this app, there will always be a row in the `Cursor`.]

We use `Cursor`'s **`getColumnIndex` method** to get the column indices for the columns in the database's `contacts` table. (We hard coded the column names in this app, but these could be implemented as `String` constants as we did for `ROW_ID` in class `AddressBook`.) This method returns `-1` if the column is not in the query result. Class `Cursor` also provides method **`getColumnIndexOrThrow`** if you prefer to get an exception when the specified column name does not exist. Lines 87–91 use `Cursor`'s **`getString` method** to retrieve the `String` values from the `Cursor`'s columns, then display these values in the corresponding `TextView`s. Lines 93–94 close the `Cursor` and this `Activity`'s connection to the database, as they're no longer needed. It's good practice to release resources like database connections when they are not being used so that other activities can use the resources.

Overriding `Activity` Methods `onCreateOptionsMenu` and `onOptionsItemSelected`

The `ViewContact` `Activity`'s menu provides options for editing the current contact and for deleting it. Method `onCreateOptionsMenu` (Fig. 10.22, lines 99–106) uses a `MenuInflater` to create the menu from the `view_contact.xml` menu resource file, which contains the **Edit Contact** and **Delete Contact** `MenuItem`s. Method `onOptionsItemSelected` (lines 109–134) uses the selected `MenuItem`'s resource ID to determine which one was selected. If it was **Edit Contact**, lines 116–126 create a new *explicit* `Intent` for the `AddEditContact` `Activity` (Section 10.5.3), add extras to the `Intent` representing this contact's information for display in the `AddEditContact` `Activity`'s `EditText`s and launch the `Activity`. If it was **Delete Contact**, line 129 calls the utility method `deleteContact` (Fig. 10.23).

```
 98      // create the Activity's menu from a menu resource XML file
 99      @Override
100      public boolean onCreateOptionsMenu(Menu menu)
101      {
102         super.onCreateOptionsMenu(menu);
103         MenuInflater inflater = getMenuInflater();
104         inflater.inflate(R.menu.view_contact_menu, menu);
105         return true;
106      } // end method onCreateOptionsMenu
107
108      // handle choice from options menu
109      @Override
110      public boolean onOptionsItemSelected(MenuItem item)
111      {
112         switch (item.getItemId()) // switch based on selected MenuItem's ID
113         {
114            case R.id.editItem:
115               // create an Intent to launch the AddEditContact Activity
116               Intent addEditContact =
117                  new Intent(this, AddEditContact.class);
118
119               // pass the selected contact's data as extras with the Intent
120               addEditContact.putExtra("row_id", rowID);
121               addEditContact.putExtra("name", nameTextView.getText());
122               addEditContact.putExtra("phone", phoneTextView.getText());
123               addEditContact.putExtra("email", emailTextView.getText());
124               addEditContact.putExtra("street", streetTextView.getText());
125               addEditContact.putExtra("city", cityTextView.getText());
126               startActivity(addEditContact); // start the Activity
127               return true;
128            case R.id.deleteItem:
129               deleteContact(); // delete the displayed contact
130               return true;
131            default:
132               return super.onOptionsItemSelected(item);
133         } // end switch
134      } // end method onOptionsItemSelected
135
```

Fig. 10.22 | Overriding methods `onCreateOptionsMenu` and `onOptionsItemSelected`.

Method `deleteContact`

Method `deleteContact` (Fig. 10.23) displays an `AlertDialog` asking the user to confirm that the currently displayed contact should be deleted, and, if so, uses an `AsyncTask` to delete it from the SQLite database. If the user clicks the **Delete** `Button` in the dialog, lines

153–154 create a new DatabaseConnector. Lines 158–173 create an AsyncTask that, when executed (line 176), passes a Long value representing the contact's row ID to the doInBackground, which then deletes the contact. Line 164 calls the DatabaseConnector's deleteContact method to perform the actual deletion. When the doInBackground completes execution, line 171 calls this Activity's finish method to return to the Activity that launched the ViewContact Activity—that is, the AddressBook Activity.

```
136    // delete a contact
137    private void deleteContact()
138    {
139       // create a new AlertDialog Builder
140       AlertDialog.Builder builder =
141          new AlertDialog.Builder(ViewContact.this);
142
143       builder.setTitle(R.string.confirmTitle); // title bar string
144       builder.setMessage(R.string.confirmMessage); // message to display
145
146       // provide an OK button that simply dismisses the dialog
147       builder.setPositiveButton(R.string.button_delete,
148          new DialogInterface.OnClickListener()
149          {
150             @Override
151             public void onClick(DialogInterface dialog, int button)
152             {
153                final DatabaseConnector databaseConnector =
154                   new DatabaseConnector(ViewContact.this);
155
156                // create an AsyncTask that deletes the contact in another
157                // thread, then calls finish after the deletion
158                AsyncTask<Long, Object, Object> deleteTask =
159                   new AsyncTask<Long, Object, Object>()
160                   {
161                      @Override
162                      protected Object doInBackground(Long... params)
163                      {
164                         databaseConnector.deleteContact(params[0]);
165                         return null;
166                      } // end method doInBackground
167
168                      @Override
169                      protected void onPostExecute(Object result)
170                      {
171                         finish(); // return to the AddressBook Activity
172                      } // end method onPostExecute
173                   }; // end new AsyncTask
174
175                // execute the AsyncTask to delete contact at rowID
176                deleteTask.execute(new Long[] { rowID });
177             } // end method onClick
178          } // end anonymous inner class
179       ); // end call to method setPositiveButton
```

Fig. 10.23 | deleteContact method of class ViewContact. (Part 1 of 2.)

```

      builder.setNegativeButton(R.string.button_cancel, null);
      builder.show(); // display the Dialog
   } // end method deleteContact
} // end class ViewContact
```

Fig. 10.23 | `deleteContact` method of class `ViewContact`. (Part 2 of 2.)

10.5.3 `AddEditContact` Subclass of `Activity`

The `AddEditContact` `Activity` (Figs. 10.24–10.27) enables the user to add a new contact or to edit an existing contact's information.

`package` Statement, `import` Statements and Instance Variables

Figure 10.24 lists the `package` statement, the `import` statements and the instance variables for class `AddEditContact`. No new classes are used in this `Activity`. Instance variable `databaseConnector` allows this `Activity` to interact with the database. Instance variable `rowID` represents the current contact being manipulated if this `Activity` was launched to allow the user to edit an existing contact. The instance variables at lines 20–24 enable us to manipulate the text in the `Activity`'s `EditText`s.

```
 1  // AddEditContact.java
 2  // Activity for adding a new entry to or
 3  // editing an existing entry in the address book.
 4  package com.deitel.addressbook;
 5
 6  import android.app.Activity;
 7  import android.app.AlertDialog;
 8  import android.os.AsyncTask;
 9  import android.os.Bundle;
10  import android.view.View;
11  import android.view.View.OnClickListener;
12  import android.widget.Button;
13  import android.widget.EditText;
14
15  public class AddEditContact extends Activity
16  {
17     private long rowID; // id of contact being edited, if any
18
19     // EditTexts for contact information
20     private EditText nameEditText;
21     private EditText phoneEditText;
22     private EditText emailEditText;
23     private EditText streetEditText;
24     private EditText cityEditText;
25
```

Fig. 10.24 | `package` statement, `import` statements and instance variables of class `AddEditContact`.

Overriding Activity *Method* onCreate

Method onCreate (Fig. 10.25) initializes the AddEditContact Activity. Lines 33–37 get the Activity's EditTexts. Next, we use Activity method getIntent to get the Intent that launched the Activity and call the Intent's getExtras method to get the Intent's Bundle of extras. When we launch the AddEditContact Activity from the AddressBook Activity, we don't add any extras to the Intent, because the user is about to specify a new contact's information. In this case, getExtras will return null. If it returns a Bundle (line 42) then the Activity was launched from the ViewContact Activity and the user has chosen to edit an existing contact. Lines 44–49 read the extras out of the Bundle by calling methods getLong (line 44) and getString, and the String data is displayed in the EditTexts for editing. Lines 53–55 register a listener for the Activity's **Save Contact** Button.

```
26     // called when the Activity is first started
27     @Override
28     public void onCreate(Bundle savedInstanceState)
29     {
30        super.onCreate(savedInstanceState); // call super's onCreate
31        setContentView(R.layout.add_contact); // inflate the UI
32
33        nameEditText = (EditText) findViewById(R.id.nameEditText);
34        emailEditText = (EditText) findViewById(R.id.emailEditText);
35        phoneEditText = (EditText) findViewById(R.id.phoneEditText);
36        streetEditText = (EditText) findViewById(R.id.streetEditText);
37        cityEditText = (EditText) findViewById(R.id.cityEditText);
38
39        Bundle extras = getIntent().getExtras(); // get Bundle of extras
40
41        // if there are extras, use them to populate the EditTexts
42        if (extras != null)
43        {
44           rowID = extras.getLong("row_id");
45           nameEditText.setText(extras.getString("name"));
46           emailEditText.setText(extras.getString("email"));
47           phoneEditText.setText(extras.getString("phone"));
48           streetEditText.setText(extras.getString("street"));
49           cityEditText.setText(extras.getString("city"));
50        } // end if
51
52        // set event listener for the Save Contact Button
53        Button saveContactButton =
54           (Button) findViewById(R.id.saveContactButton);
55        saveContactButton.setOnClickListener(saveContactButtonClicked);
56     } // end method onCreate
57
```

Fig. 10.25 | Overriding Activity methods onCreate and onPause.

OnClickListener *to Process* Save Contact Button *Events*

When the user touches the **Save Contact** Button in the AddEditContact Activity, the saveContactButtonClicked OnClickListener (Fig. 10.26) executes. To save a contact, the user must enter at least the contact's name. Method onClick ensures that the length of the name is greater than 0 characters (line 64) and, if so, creates and executes an AsyncTask to perform the save operation. Method doInBackground (lines 69–74) calls saveContact (Fig. 10.27) to save the contact into the database. Method onPostExecute (lines 76–80) calls finish to terminate this Activity and return to the launching Activity (either AddressBook or ViewContact). If the nameEditText is empty, lines 89–96 show an AlertDialog telling the user that a contact name must be provided to save the contact.

```
58     // responds to event generated when user clicks the Done Button
59     OnClickListener saveContactButtonClicked = new OnClickListener()
60     {
61        @Override
62        public void onClick(View v)
63        {
64           if (nameEditText.getText().length() != 0)
65           {
66              AsyncTask<Object, Object, Object> saveContactTask =
67                 new AsyncTask<Object, Object, Object>()
68                 {
69                    @Override
70                    protected Object doInBackground(Object... params)
71                    {
72                       saveContact(); // save contact to the database
73                       return null;
74                    } // end method doInBackground
75
76                    @Override
77                    protected void onPostExecute(Object result)
78                    {
79                       finish(); // return to the previous Activity
80                    } // end method onPostExecute
81                 }; // end AsyncTask
82
83              // save the contact to the database using a separate thread
84              saveContactTask.execute((Object[]) null);
85           } // end if
86           else
87           {
88              // create a new AlertDialog Builder
89              AlertDialog.Builder builder =
90                 new AlertDialog.Builder(AddEditContact.this);
91
92              // set dialog title & message, and provide Button to dismiss
93              builder.setTitle(R.string.errorTitle);
94              builder.setMessage(R.string.errorMessage);
95              builder.setPositiveButton(R.string.errorButton, null);
96              builder.show(); // display the Dialog
97           } // end else
98        } // end method onClick
99     }; // end OnClickListener saveContactButtonClicked
100
```

Fig. 10.26 | OnClickListener doneButtonClicked responds to the events of the doneButton.

`saveContact` Method

The `saveContact` method (Fig. 10.27) saves the information in this `Activity`'s `EditTexts`. First, line 105 creates the `DatabaseConnector` object, then we check whether the `Intent` that launched this `Activity` had any extras. If not, this is a new contact, so lines 110–115 get the `Strings` from the `Activity`'s `EditTexts` and pass them to the `DatabaseConnector` object's `insertContact` method to create the new contacts. If there are extras for the `Intent` that launched this `Activity`, then an existing contact is being updated. In this case, we get the `Strings` from the `Activity`'s `EditTexts` and pass them to the `DatabaseConnector` object's `updateContact` method, using the `rowID` to indicate which record to update. `DatabaseConnector` methods `insertContact` and `updateContact` each handle the opening and closing of the database,

```
101    // saves contact information to the database
102    private void saveContact()
103    {
104       // get DatabaseConnector to interact with the SQLite database
105       DatabaseConnector databaseConnector = new DatabaseConnector(this);
106
107       if (getIntent().getExtras() == null)
108       {
109          // insert the contact information into the database
110          databaseConnector.insertContact(
111             nameEditText.getText().toString(),
112             emailEditText.getText().toString(),
113             phoneEditText.getText().toString(),
114             streetEditText.getText().toString(),
115             cityEditText.getText().toString());
116       } // end if
117       else
118       {
119          databaseConnector.updateContact(rowID,
120             nameEditText.getText().toString(),
121             emailEditText.getText().toString(),
122             phoneEditText.getText().toString(),
123             streetEditText.getText().toString(),
124             cityEditText.getText().toString());
125       } // end else
126    } // end class saveContact
127 } // end class AddEditContact
```

Fig. 10.27 | `saveContact` method of class `AddEditContact`.

10.5.4 DatabaseConnector Utility Class

The `DatabaseConnector` utility class (Figs. 10.28–10.31) manages this app's interactions with SQLite for creating and manipulating the `UserContacts` database, which contains one table named `contacts`.

`package` Statement, `import` Statements and Fields

Figure 10.28 lists class `DatabaseConnector`'s `package` statement, `import` statements and fields. We've highlighted the `import` statements for the new classes and interfaces discussed in Section 10.3. The `String` constant `DATABASE_NAME` (line 16) specifies the name of the database that will be created or opened. *Database names must be unique within a specific app but need not be unique across apps.* A `SQLiteDatabase` object (line 17) provides read/write access to a SQLite database. The `DatabaseOpenHelper` (line 18) is a `private` nested class that extends abstract class `SQLiteOpenHelper`—such a class is used to manage creating, opening and upgrading databases (perhaps to modify a database's structure). We discuss `SQLOpenHelper` in more detail in Fig. 10.31.

```
 1  // DatabaseConnector.java
 2  // Provides easy connection and creation of UserContacts database.
 3  package com.deitel.addressbook;
 4
 5  import android.content.ContentValues;
 6  import android.content.Context;
 7  import android.database.Cursor;
 8  import android.database.SQLException;
 9  import android.database.sqlite.SQLiteDatabase;
10  import android.database.sqlite.SQLiteOpenHelper;
11  import android.database.sqlite.SQLiteDatabase.CursorFactory;
12
13  public class DatabaseConnector
14  {
15     // database name
16     private static final String DATABASE_NAME = "UserContacts";
17     private SQLiteDatabase database; // database object
18     private DatabaseOpenHelper databaseOpenHelper; // database helper
19
```

Fig. 10.28 | `package` statement, `import` statements and instance variables of utility class `DatabaseConnector`.

Constructor and Methods `open` *and* `close` *for Class* `DatabaseConnector`

`DatabaseConnection`'s constructor (Fig. 10.29, lines 21–26) creates a new object of class `DatabaseOpenHelper` (Fig. 10.31), which will be used to open or create the database. We discuss the details of the `DatabaseOpenHelper` constructor in Fig. 10.31. The `open` method (lines 29–33) attempts to establish a connection to the database and throws a `SQLException` if the connection attempt fails. Method **`getWritableDatabase`** (line 32), which is inherited from `SQLiteOpenHelper`, returns a `SQLiteDatabase` object. If the database has not yet been created, this method will create it; otherwise, the method will open it. Once the database is opened successfully, it will be *cached* by the operating system to improve the performance of future database interactions. The `close` method (lines 36–40) closes the database connection by calling the inherited `SQLiteOpenHelper` method **`close`**.

```
20     // public constructor for DatabaseConnector
21     public DatabaseConnector(Context context)
22     {
```

Fig. 10.29 | Constructor, `open` method and `close` method. (Part 1 of 2.)

```
      // create a new DatabaseOpenHelper
      databaseOpenHelper =
         new DatabaseOpenHelper(context, DATABASE_NAME, null, 1);
   } // end DatabaseConnector constructor

   // open the database connection
   public void open() throws SQLException
   {
      // create or open a database for reading/writing
      database = databaseOpenHelper.getWritableDatabase();
   } // end method open

   // close the database connection
   public void close()
   {
      if (database != null)
         database.close(); // close the database connection
   } // end method close

```

Fig. 10.29 | Constructor, `open` method and `close` method. (Part 2 of 2.)

Methods `insertContact`*,* `updateContact`*,* `getAllContacts`*,* `getOneContact` *and* `deleteContact`

Method `insertContact` (Fig. 10.30, lines 43–56) inserts a new contact with the given information into the database. We first put each piece of contact information into a new **`ContentValues`** object (lines 46–51), which maintains a map of key–value pairs—the database's column names are the keys. Lines 53–55 open the database, insert the new contact and close the database. `SQLiteDatabase`'s **`insert` method** (line 54) inserts the values from the given `ContentValues` into the table specified as the first argument—the `"contacts"` table in this case. The second parameter of this method, which is not used in this app, is named `nullColumnHack` and is needed because *SQLite does not support inserting a completely empty row into table*—this would be the equivalent of passing an empty `ContentValues` object to `insert`. Instead of making it illegal to pass an empty `ContentValues` to the method, the `nullColumnHack` parameter is used to identify a column that accepts `NULL` values.

```
42    // inserts a new contact in the database
43    public void insertContact(String name, String email, String phone,
44       String state, String city)
45    {
46       ContentValues newContact = new ContentValues();
47       newContact.put("name", name);
48       newContact.put("email", email);
49       newContact.put("phone", phone);
50       newContact.put("street", state);
51       newContact.put("city", city);
52
```

Fig. 10.30 | Methods `insertContact`, `updateContact`, `getAllContacts`, `getOneContact` and `deleteContact`. (Part 1 of 2.)

```
53         open(); // open the database
54         database.insert("contacts", null, newContact);
55         close(); // close the database
56      } // end method insertContact
57
58      // inserts a new contact in the database
59      public void updateContact(long id, String name, String email,
60         String phone, String state, String city)
61      {
62         ContentValues editContact = new ContentValues();
63         editContact.put("name", name);
64         editContact.put("email", email);
65         editContact.put("phone", phone);
66         editContact.put("street", state);
67         editContact.put("city", city);
68
69         open(); // open the database
70         database.update("contacts", editContact, "_id=" + id, null);
71         close(); // close the database
72      } // end method updateContact
73
74      // return a Cursor with all contact information in the database
75      public Cursor getAllContacts()
76      {
77         return database.query("contacts", new String[] {"_id", "name"},
78            null, null, null, null, "name");
79      } // end method getAllContacts
80
81      // get a Cursor containing all information about the contact specified
82      // by the given id
83      public Cursor getOneContact(long id)
84      {
85         return database.query(
86            "contacts", null, "_id='" + id, null, null, null, null);
87      } // end method getOnContact
88
89      // delete the contact specified by the given String name
90      public void deleteContact(long id)
91      {
92         open(); // open the database
93         database.delete("contacts", "_id=" + id, null);
94         close(); // close the database
95      } // end method deleteContact
96
```

Fig. 10.30 | Methods `insertContact`, `updateContact`, `getAllContacts`, `getOneContact` and `deleteContact`. (Part 2 of 2.)

Method `updateContact` (lines 59–72) is similar to method `insertContact`, except that it calls `SQLiteDatabase`'s **`update` method** (line 70) to update an existing contact. The `update` method's third argument represents a SQL `WHERE` clause (without the keyword `WHERE`) that specifies which record(s) to update. In this case, we use the record's row ID to update a specific contact.

Method `getAllContacts` (lines 75–79) uses `SqLiteDatabase`'s **`query` method** (lines 77–78) to retrieve a `Cursor` that provides access to the IDs and names of all the contacts in the database. The arguments are:

- the name of the table to query
- a `String` array of the column names to return (the `_id` and `name` columns here)—`null` returns all columns in the table, which is generally a poor programming practice, because to conserve memory, processor time and battery power, you should obtain only the data you need
- a SQL `WHERE` clause (without the keyword `WHERE`), or `null` to return all rows
- a `String` array of arguments to be substituted into the `WHERE` clause wherever `?` is used as a placeholder for an argument value, or `null` if there are no arguments in the `WHERE` clause
- a SQL `GROUP BY` clause (without the keywords `GROUP BY`), or `null` if you don't want to group the results
- a SQL `HAVING` clause (without the keyword `HAVING`) to specify which groups from the `GROUP BY` clause to include in the results—`null` is required if the `GROUP BY` clause is `null`
- a SQL `ORDER BY` clause (without the keywords `ORDER BY`) to specify the order of the results, or `null` if you don't wish to specify the order.

The `Cursor` returned by method `query` contains all the table rows that match the method's arguments—the so-called *result set*. The `Cursor` is positioned *before* the first row of the result set—`Cursor`'s various `move` methods can be used to move the `Cursor` through the result set for processing.

Method `getOneContact` (lines 83–87) also uses `SqLiteDatabase`'s `query` method to query the database. In this case, we retrieve all the columns in the database for the contact with the specified ID.

Method `deleteContact` (lines 90–95) uses `SqLiteDatabase`'s **`delete` method** (line 93) to delete a contact from the database. In this case, we retrieve all the columns in the database for the contact with the specified ID. The three arguments are the database table from which to delete the record, the `WHERE` clause (without the keyword `WHERE`) and, if the `WHERE` clause has arguments, a `String` array of values to substitute into the `WHERE` clause (`null` in our case).

`private` Nested Class `DatabaseOpenHelper` That Extends `SQLiteOpenHelper`

The `private` nested class `DatabaseOpenHelper` (Fig. 10.31) extends abstract class `SQLiteOpenHelper`, which helps apps create databases and manage version changes. The constructor (lines 100–104) simply calls the superclass constructor, which requires four arguments:

- the `Context` in which the database is being created or opened,
- the database name—this can be `null` if you wish to use an in-memory database,
- the `CursorFactory` to use—`null` indicates that you wish to use the default SQLite `CursorFactory` (typically for most apps) and
- the database version number (starting from 1).

You must override this class's abstract methods onCreate and onUpgrade. If the database does not yet exist, the DatabaseOpenHelper's **onCreate method** will be called to create it. If you supply a newer version number than the database version currently stored on the device, the DatabaseOpenHelper's **onUpgrade method** will be called to upgrade the database to the new version (perhaps to add tables or to add columns to an existing table).

```
 97    private class DatabaseOpenHelper extends SQLiteOpenHelper
 98    {
 99       // public constructor
100       public DatabaseOpenHelper(Context context, String name,
101          CursorFactory factory, int version)
102       {
103          super(context, name, factory, version);
104       } // end DatabaseOpenHelper constructor
105
106       // creates the contacts table when the database is created
107       @Override
108       public void onCreate(SQLiteDatabase db)
109       {
110          // query to create a new table named contacts
111          String createQuery = "CREATE TABLE contacts" +
112             "(_id integer primary key autoincrement," +
113             "name TEXT, email TEXT, phone TEXT," +
114             "street TEXT, city TEXT);";
115
116          db.execSQL(createQuery); // execute the query
117       } // end method onCreate
118
119       @Override
120       public void onUpgrade(SQLiteDatabase db, int oldVersion,
121          int newVersion)
122       {
123       } // end method onUpgrade
124    } // end class DatabaseOpenHelper
125 } // end class DatabaseConnector
```

Fig. 10.31 | SQLiteOpenHelper class DatabaseOpenHelper.

The onCreate method (lines 107–117) specifies the table to create with the SQL CREATE TABLE command, which is defined as a String (lines 111–114). In this case, the contacts table contains an integer primary key field (_id) that is auto-incremented, and text fields for all the other columns. Line 116 uses SQLiteDatabase's **execSQL** method to execute the CREATE TABLE command. Since we don't need to upgrade the database, we simply override method onUpgrade with an empty body. As of Android 3.0, class SQLiteOpenHelper also provides an **onDowngrade method** that can be used to downgrade a database when the currently stored version has a higher version number than the one requested in the call to class SQLiteOpenHelper's constructor. Downgrading might be used to revert the database back to a prior version with fewer columns in a table or fewer tables in the database—perhaps to fix a bug in the app.

All the SQLiteDatabase methods we used in class DatabaseConnector have corresponding methods which perform the same operations but throw exceptions on failure, as opposed to simply returning -1 (e.g., insertOrThrow vs. insert). These methods are interchangeable, allowing you to decide how to deal with database read and write errors.

10.6 Wrap-Up

In this chapter, you created an **Address Book** app that enables users to add, view, edit and delete contact information that's stored in a SQLite database. You learned that every Activity in an app must be described in the app's AndroidManifest.xml file.

You defined common GUI component attribute–value pairs as XML style resources, then applied the styles to all components that share those values by using the components' style attribute. You added a border to a TextView by specifying a Drawable as the value for the TextView's android:background attribute and you created a custom Drawable using an XML representation of a shape.

You used XML menu resources to define the app's MenuItems and programmatically inflated them using an Activity's MenuInflater. You also used Android standard icons to enhance the visual appearance of the menu items.

When an Activity's primary task is to display a scrollable list of items, you learned that you can extend class ListActivity to create an Activity that displays a ListView in its default layout. You used this to display the contacts stored in the app's database. You also saw that a ListView is a subclass of AdapterView, which allows a component to be bound to a data source, and you used a CursorAdapter to display the results of a database query in main Activity's ListView.

You used explicit Intents to launch new activities that handled tasks such as adding a contact, editing an existing contact and deleting an existing contact. You also learned how to terminate a launched activity to return to the prior one using the Activity's finish method.

You used a subclass of SQLiteOpenHelper to simplify creating the database and to obtain a SQLiteDatabase object for manipulating a database's contents. You processed query results via a Cursor. You used subclasses of AsyncTask to perform database tasks outside the GUI thread and return results to the GUI thread. This allowed you to take advantage of Android's threading capabilities without directly creating and manipulating threads.

In Chapter 11, we present the **Route Tracker** app, which uses GPS technology to track the user's location and draws that location on a street map overlaid on a satellite image. The app uses a MapView to interact with the Google Maps web services and display the maps, and uses an Overlay to display the user's location. The app also receives GPS data and direction information from the Android location services and sensors.