Chapter 5

# Reminders Lab: Part 1

By now you are familiar with the basics of creating a new project, programming, and refactoring. It is time to create an Android application, otherwise known as an app. This chapter introduces the first of four lab projects. These labs are intended to familiarize you with using Android Studio in the context of developing an app. In this project, you will develop an app to manage a list of items you want to remember. The core functionality will allow you to create and delete reminders and flag certain reminders as important. An important item will be emphasized by an orange tab to the left of the reminder's text. The app will incorporate an action bar menu, context menus, a local database for persistence, and multiple selection on devices that support multiple selection.

Figure 5-1 illustrates the completed app running on the emulator. This example introduces you to Android fundamentals and you will also learn how to persist data by using the built-in SQLite database. Don't worry if some of the topics are unfamiliar; later chapters cover those topics in greater detail.

> **Note** We invite you to clone this project using Git in order to follow along, though you will be recreating this project with its own Git repository from scratch. If you do not have Git installed on your computer, see Chapter 7. Open a Git-bash session in Windows (or a terminal in Mac or Linux) and navigate to `C:\androidBook\reference\` (If you do not have a reference directory, create one. On Mac navigate to /your-labs-parent-dir/reference/) and issue the following git command: git clone `https://bitbucket.org/csgerber/reminders.git Reminders`.

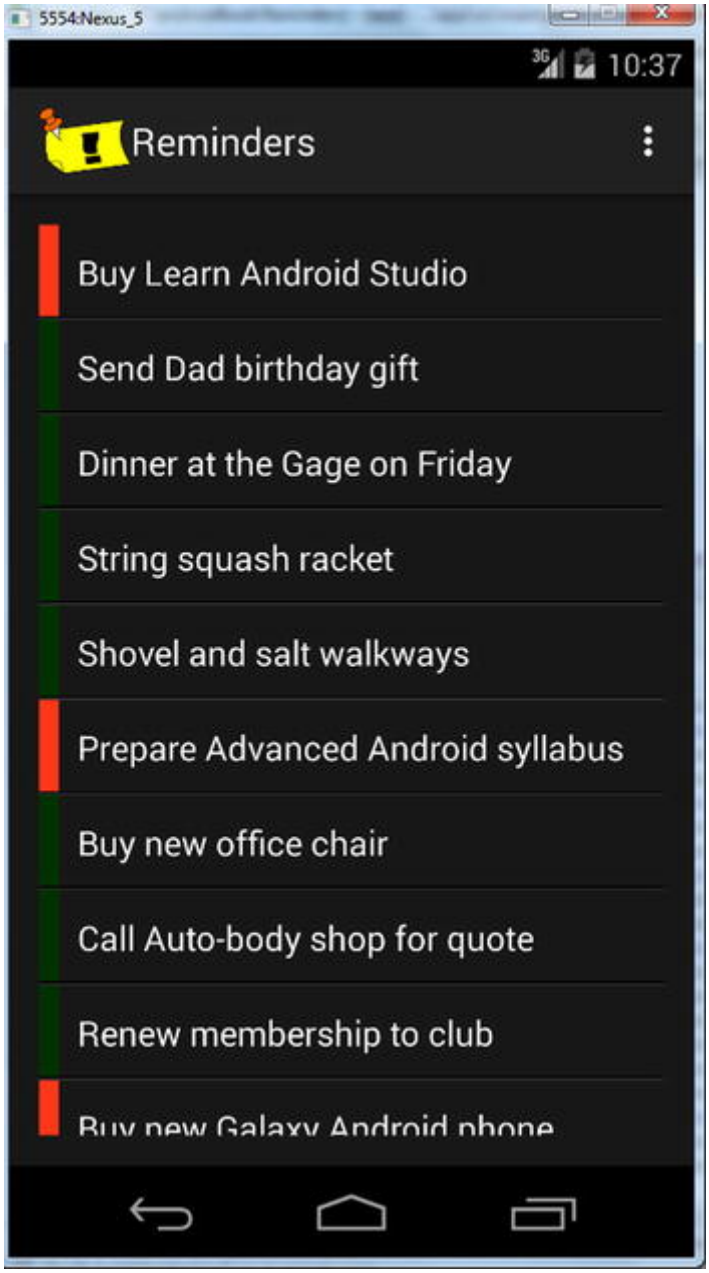


***Figure 5-1.*** *The completed app interface*

To operate the Reminders app, you can use the overflow menu of the Action Bar. Tapping the overflow button, which looks like three vertical dots on the right side of the menu bar, opens a menu with two options as shown in Figure 5-2: New Reminder, and Exit. Tapping New Reminder opens a dialog box as shown in Figure 5-3. In the dialog box, you can add text for your new reminder and then tap Commit to add it to the list. Tapping Exit simply quits the app.

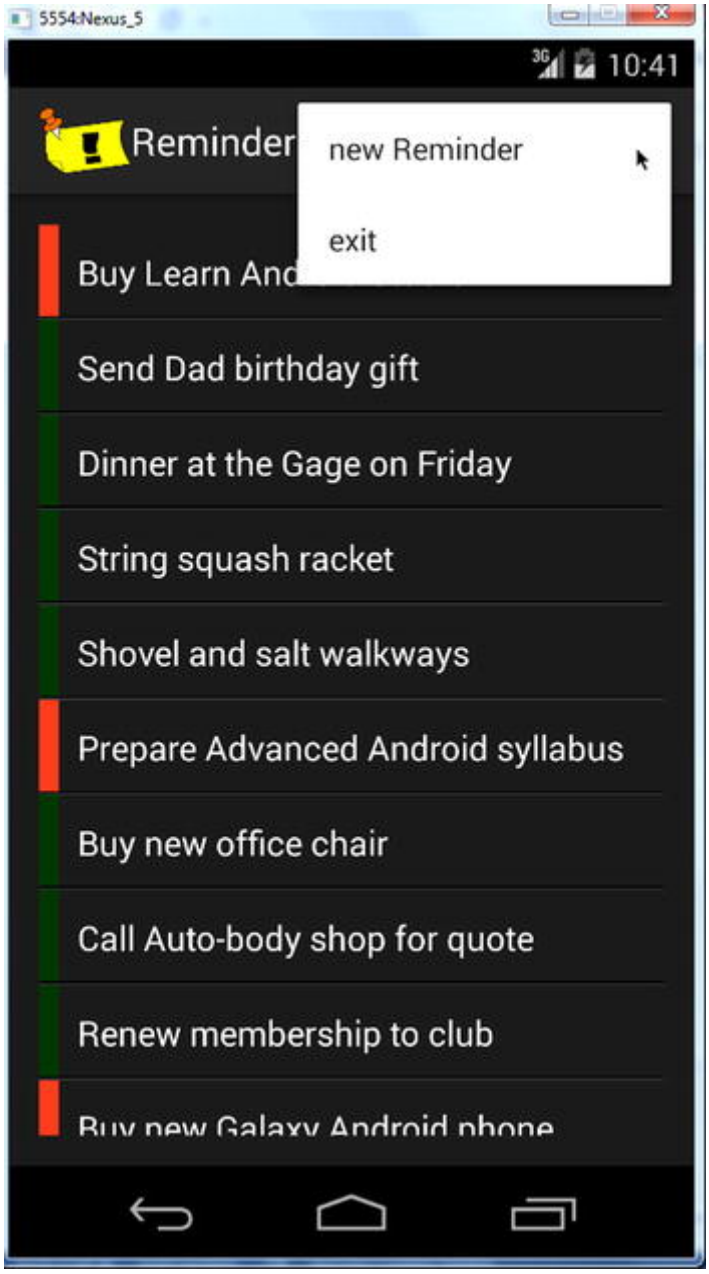


***Figure 5-2.*** *App interface with overflow menu activated*

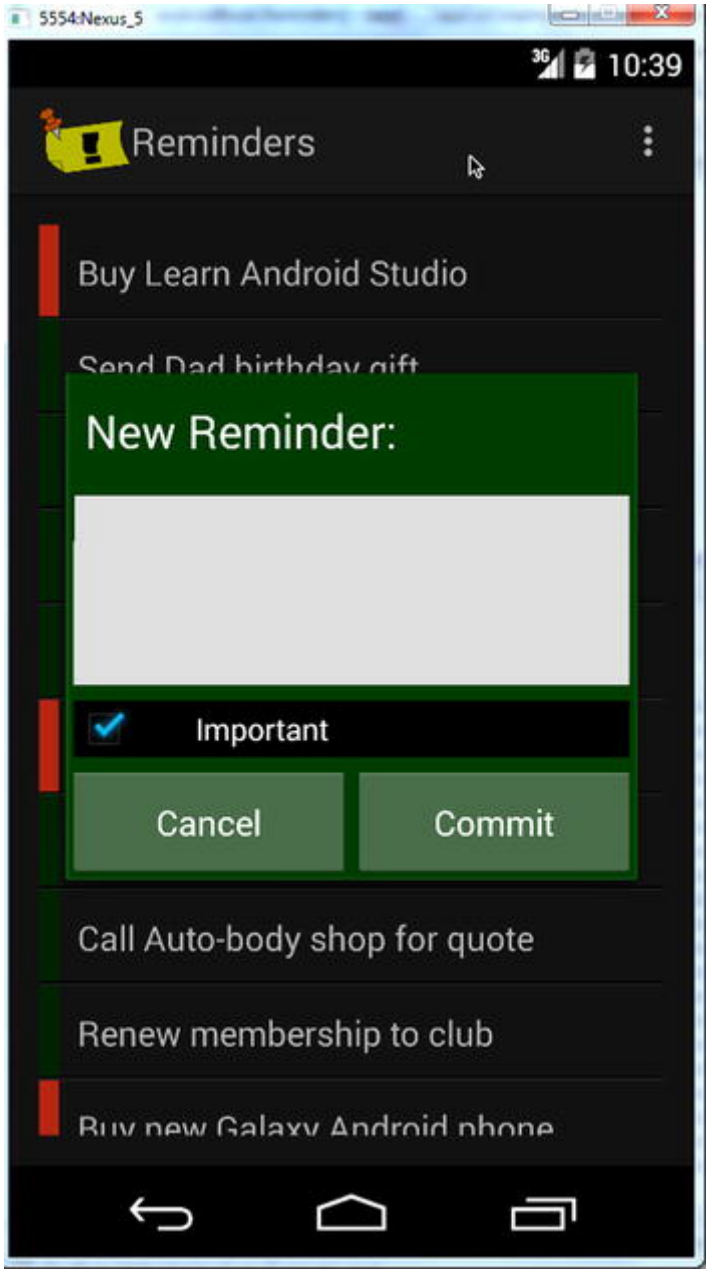


***Figure 5-3.*** *New Reminder dialog box*

Tapping any reminder in the list opens a context menu with two options, shown in Figure 5-4: Edit Reminder and Delete Reminder. Tapping Edit Reminder from the context menu opens the Edit Reminder pop-up dialog box shown in Figure 5-5, where you can change the text of the reminder. Tapping Delete Reminder from the context menu deletes the reminder from the list.

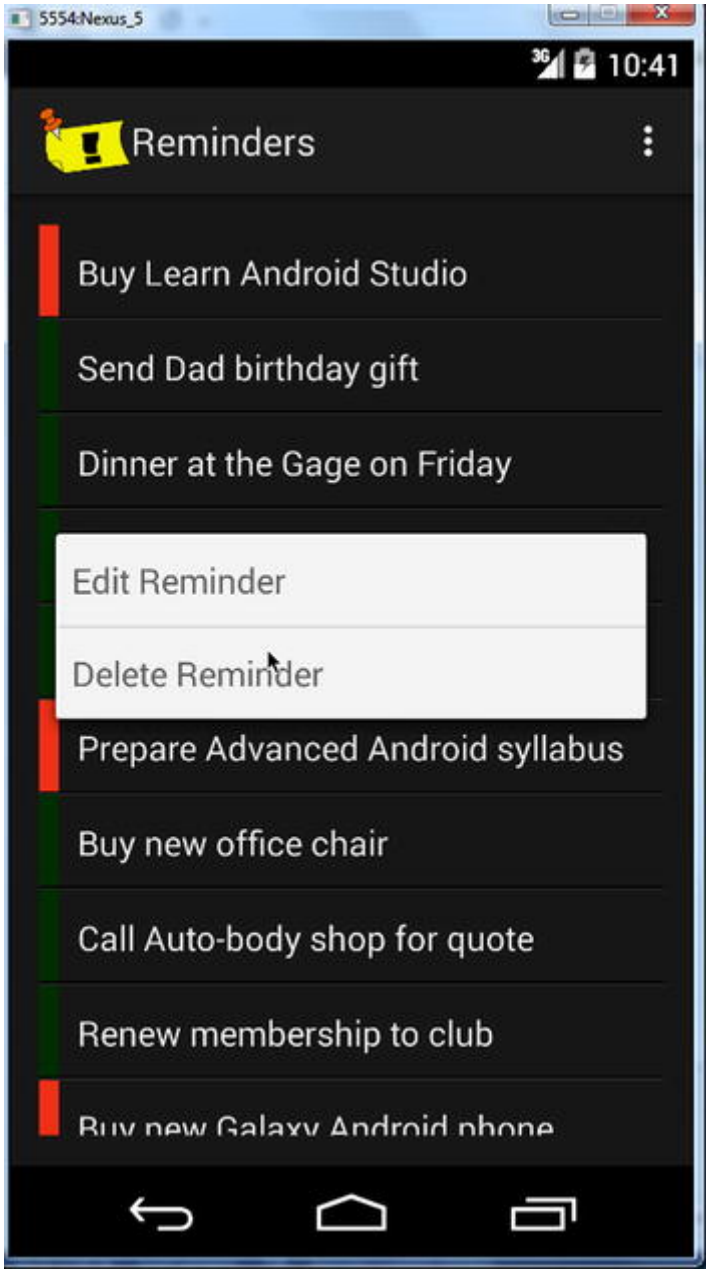


***Figure 5-4.*** *Context menu*

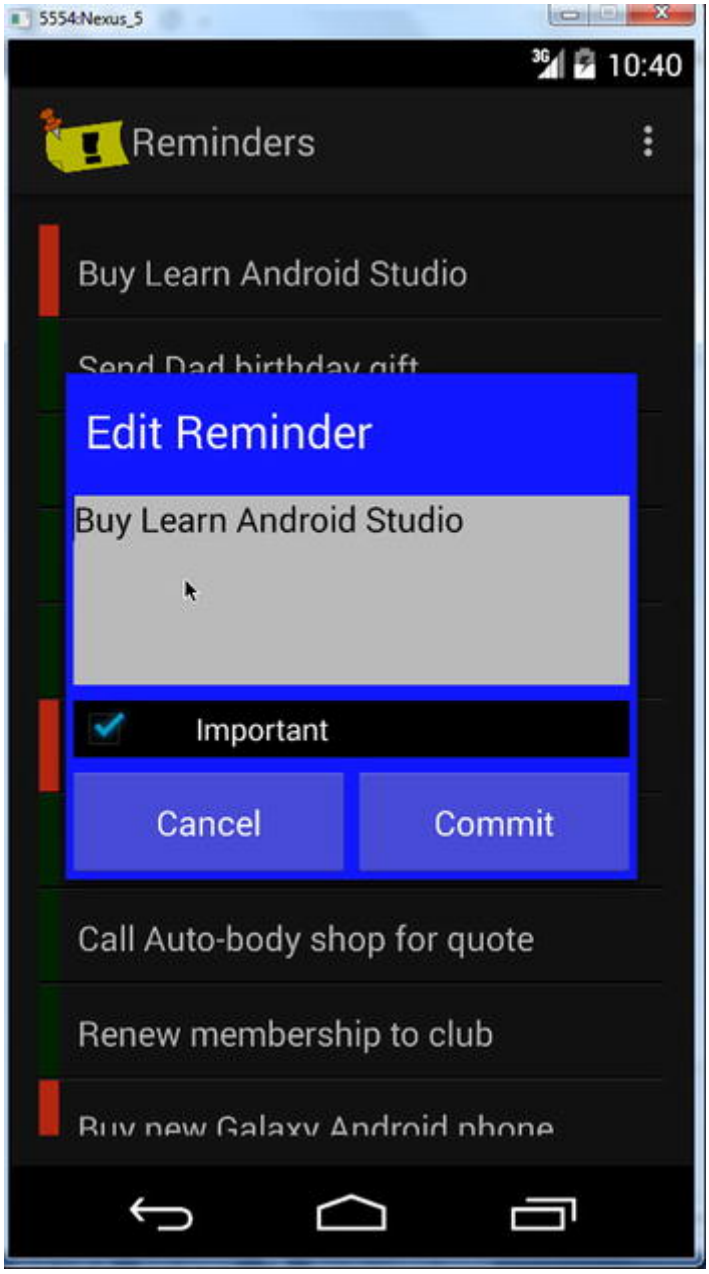


***Figure 5-5.*** *Edit Reminder dialog box*

# Starting a New Project

Start a new project in Android Studio by using the New Project Wizard as explained in Chapter 1. Enter **Reminders** as the application name, set the company domain to `gerber.apress.com`, and choose the Blank Activity template. Save this project under the path `C:\androidBook\Reminders`. It's a good idea to keep all of your lab projects in a common folder such as `C:\androidBook` (or use `~/androidBook` for Mac/Linux) for consistency with our examples. On the next page of the wizard, select Phone and Tablet and set the Minimum SDK to API 8: Android 2.2 (Froyo). By setting your min API level to 8, you are making your app available to more than 99% of the Android market. Click the Next button, choose the Blank Activity from the available templates, and click Next again.

Set the activity name to `RemindersActivity` and then click Finish, as shown in Figure 5-6.

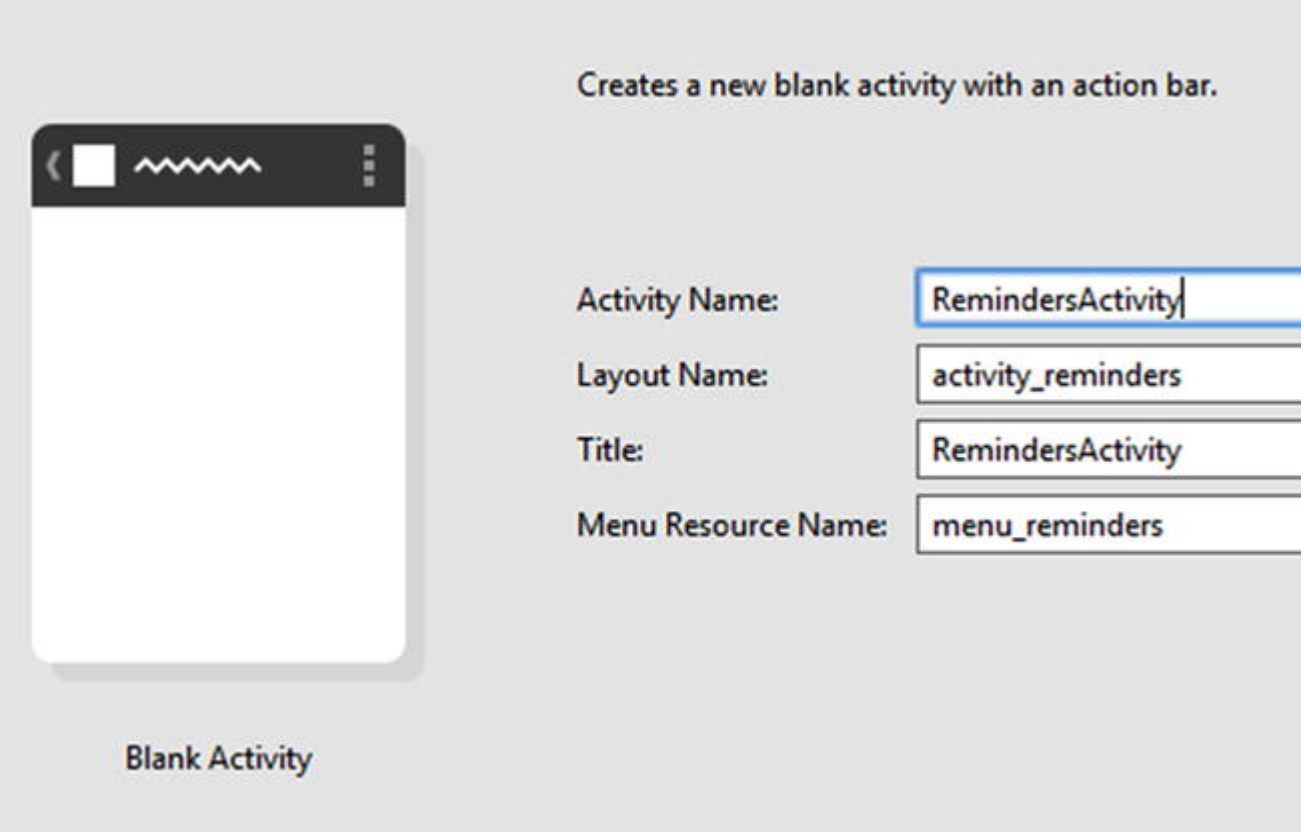


***Figure 5-6.*** *Entering an activity name*

Android Studio displays `activity_reminders.xml` in Design mode. The `activity_reminders.xml` file is the layout for your main activity, as shown in Figure 5-7. As discussed in Chapter 1, the project should run on either an emulator or a device at this point. Feel free to connect your device or launch your emulator and the run the project to try it out.

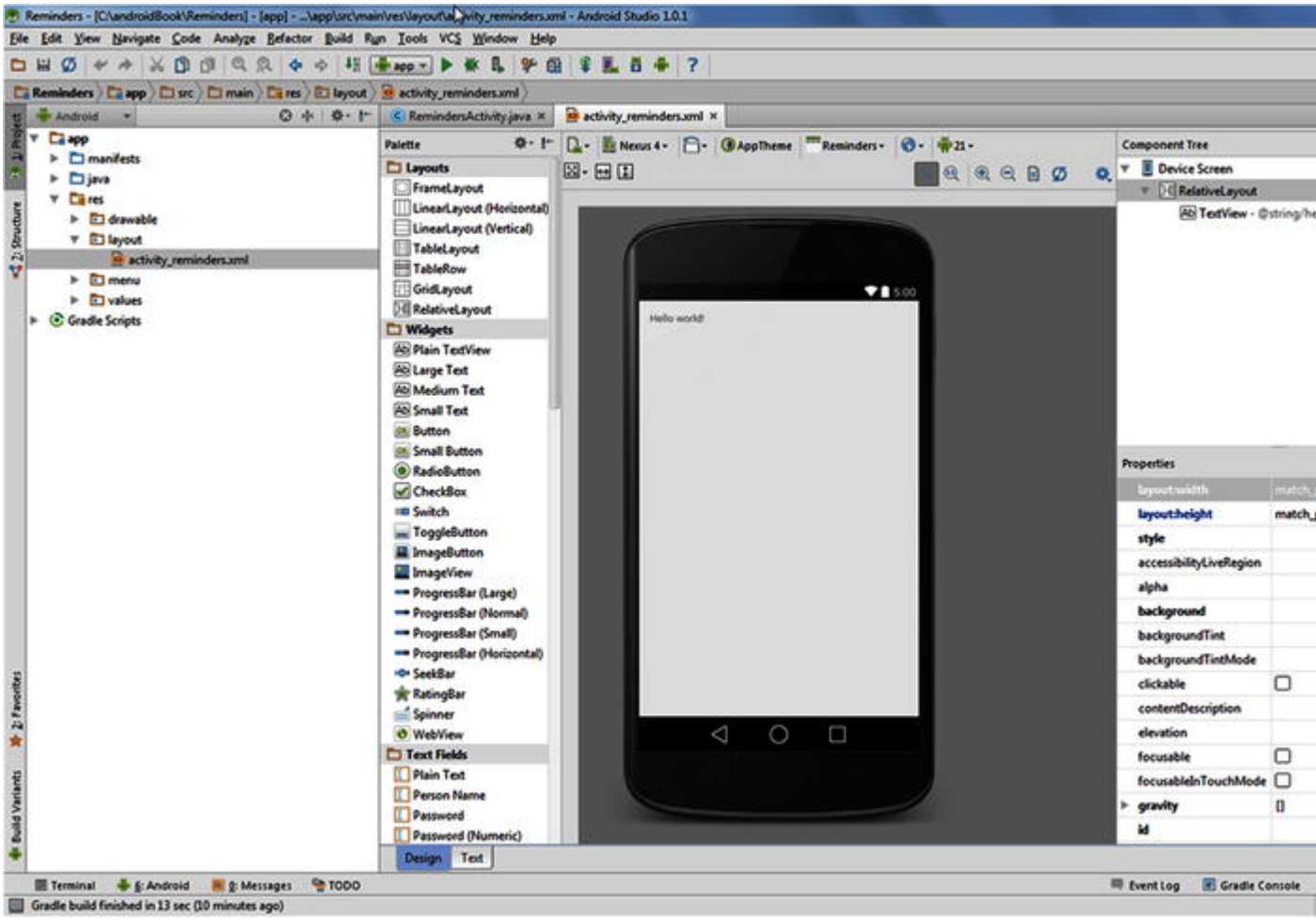


***Figure 5-7.*** *Design mode for activity_reminders*

# Initializing the Git Repository

Your first step after creating a new project should be to manage the source code with version control. All the labs is this book use Git, a popular version-control system that works seamlessly with Android Studio and is available online for free. Chapter 7 explores Git and version control more thoroughly.

If you do not already have Git installed on your computer, please refer to the section entitled Installing Git in Chapter 7. Choose VCS ➤Import into Version Control ➤Create Git Repository from the main menu. (In the Apple OS, choose VCS ➤VCS Operations ➤Create Git Repository.) Figures 5-8 and 5-9 demonstrate this flow.

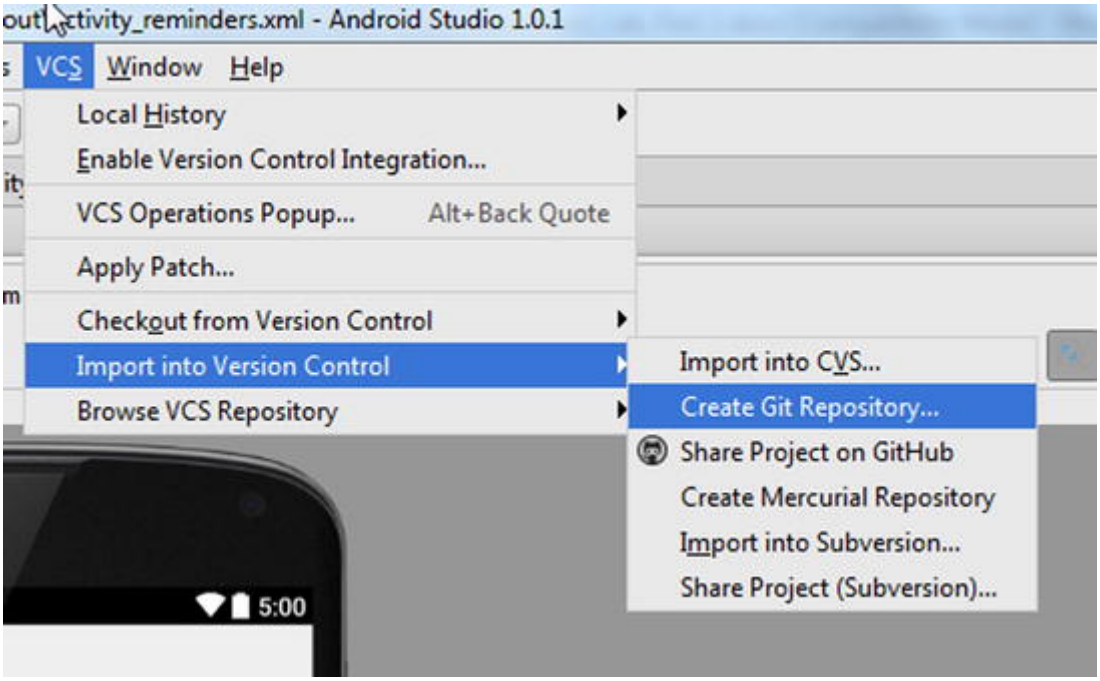


***Figure 5-8.*** *Creating the Git repository*

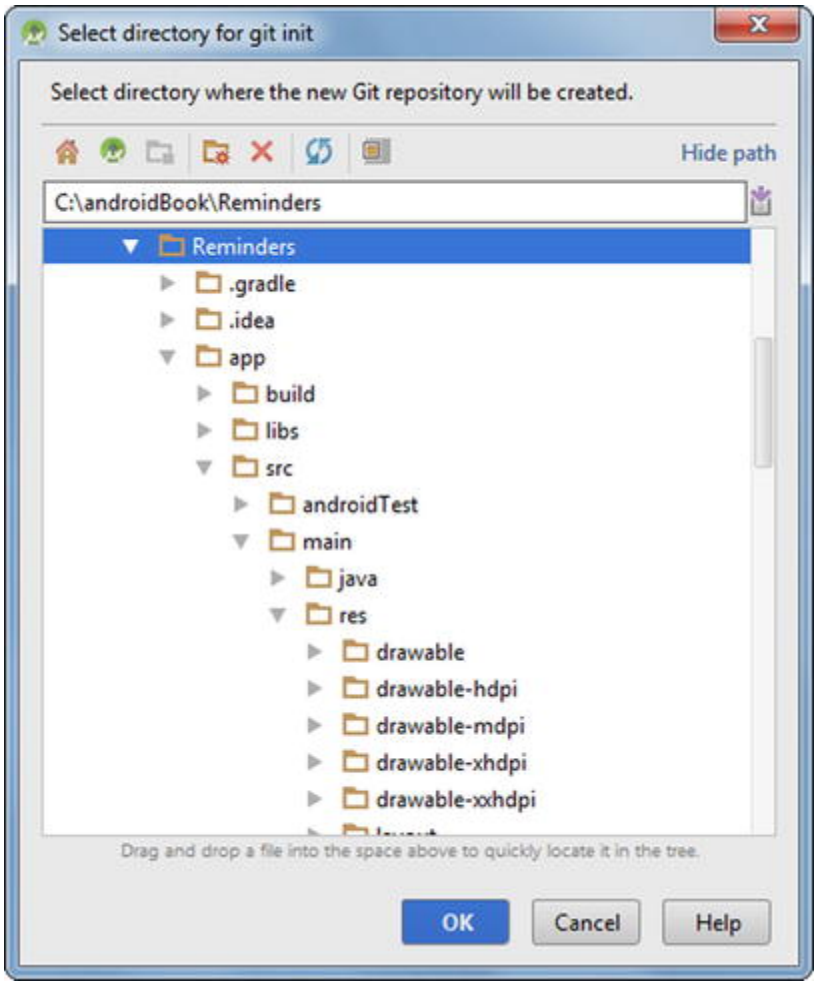


***Figure 5-9.*** *Selecting the root directory for the Git repository*

When prompted to select the directory for Git init, make sure that the Git project will be initialized in the root project directory (again, called `Reminders` in this example). Click OK.

You will notice that most of the files located in the Project tool window have turned brown, which means that they are being tracked by Git but have not yet been added to the Git repository nor are they scheduled to be added. Once your project is under Git's control, Android Studio uses a coloring scheme to indicate the status of files as they are created, modified, or deleted. This coloring scheme will be explained in more detail as we progress though you can research this topic in more detail here: `jetbrains.com/idea/help/file-status-highlights.html`.

Click the Changes tool button located along the bottom margin to toggle open the Changes tool window and expand the leaf labeled Unversioned Files. This will show all files that are being tracked. To add them, select the Unversioned Files leaf and press Ctrl+Alt+A | Cmd+Alt+A or right-click the Unversioned Files leaf and choose Git ➤Add. The brown files should have turned green, which means that they have been staged in Git and are now ready to be committed.

Press Ctrl+K | Cmd+K to invoke the Commit Changes dialog box. *Committing* files is the process of recording project changes to the Git version control system. As shown in Figure 5-10, the Author drop-down menu is used to override the current default committer. You should leave the Author field blank, and Android Studio will simply use the defaults you initially set during your Git installation. Deselect all check-box options in the Before Commit section. Put the following message in the Commit Message field: **Initial commit using new project wizard**. Click the Commit button and select Commit again from the drop-down items.

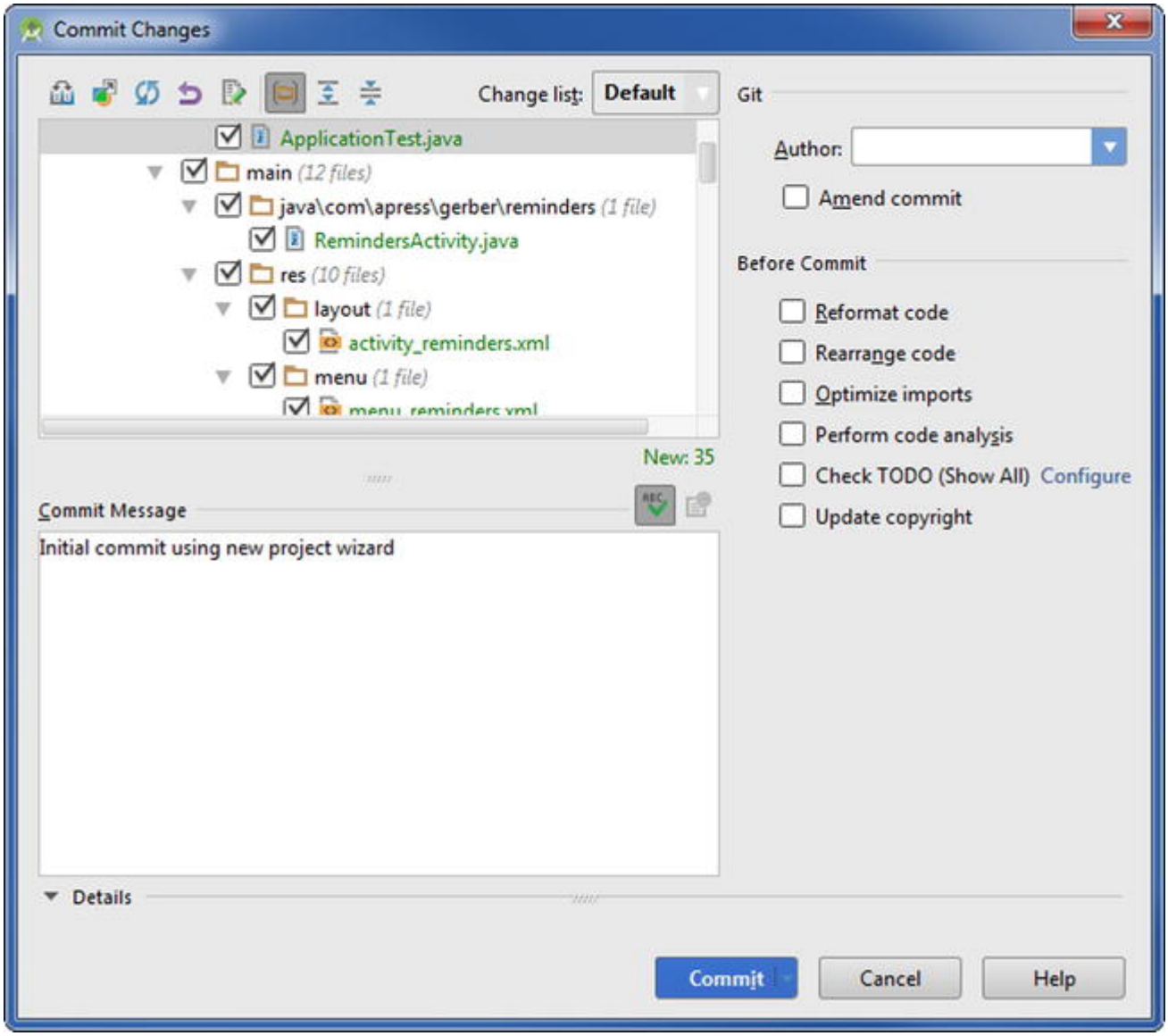


***Figure 5-10.*** *Committing changes to Git*

By default, the Project tool window should be open. The Project tool window organizes your project in different ways, depending on which view is selected in the mode drop-down menu at the top of the window. By default, the drop-down menu is set to Android view, which organizes the files according to their purpose and has nothing to do with the way the files are organized on your computer's operating system. As you explore the Projects tool window, you will notice three folders under the app folder: `manifests`, `java`, and `res`. The `manifests` folder is where your Android manifest files can be found. The `java` folder is where your Java source files may be found. The `res` folder holds all of your Android resource files. The resources located under the `res` directory may be XML files, images, sounds, and other assets that help define the appearance and UI experience of your app. Once you've had the opportunity to explore Android view, we recommend switching to Project view which is more intuitive because it maps directly to the file structure on your computer.

> **Note:** If you have worked with other IDEs or older beta versions of Android Studio, you will notice the introduction of the Android and Package views in the Project tool window since the release of Android Studio.

# Building the User Interface

By default, Android Studio opens the XML layout file associated with the main activity in a new tab of the Editor and sets its mode to Design, so the Visual Designer is typically the first thing you see in your new project. The Visual Designer lets you edit the visual layout of your app. In the middle of the screen is the Preview Pane. The Preview Pane displays a visual representation of an Android device while rendering the results of the layout you are currently editing. This representation can be controlled by using the preview layout controls across the top of the screen. These controls adjust the preview and can be used to select different (or multiple) flavors of Android devices, from smartphones to tablets or wearables. You can also change the theme associated with your layout description. On the left side of the screen, you'll find the Control palette. It contains various controls and widgets that can be dragged and placed onto the stage, which is a visual representation of the device. The right side of the IDE contains a component tree that shows the hierarchy of components described in your layout. The layout uses XML. As you make changes in the Visual Designer, these changes are updated in XML. You can click the Design and Text tabs to toggle between visual- and text-editing modes. Figure 5-11 identifies several key areas of the Visual Designer.

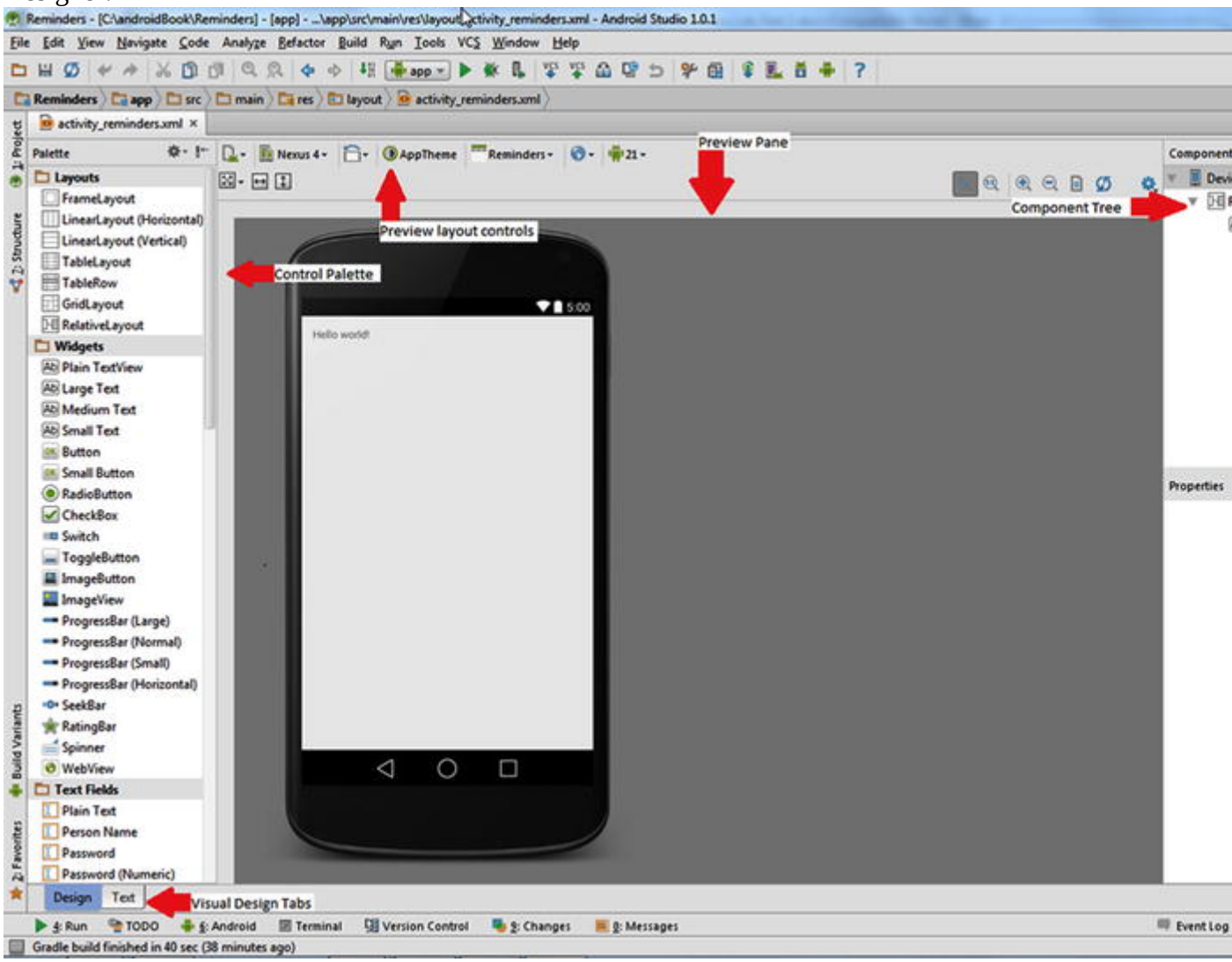


***Figure 5-11.*** *The Visual Designer layout*

# Working with the Visual Designer

Let's start by creating a list of reminders. Click the Hello World `TextView` control on the stage and then press Delete to remove it. Find the `ListView` control in the palette and drag it onto the stage. As you drag, the IDE will display various measurement and alignment guidelines to help you position the control which will tend to snap to the edges as you drag close to them. Drop the `ListView` so that it aligns with the top of the screen. You can position it either at the top-left or the top-center. After it is positioned, find the Properties view on the lower-right side of the Editor. Set the `id` property to `reminders_list_view`. The `id` property is a name you can give to controls that allows you to reference them programmatically in Java code; and this is how we will refer to the `ListView` later when we modify the Java source code. Change the `layout:width` property in the Properties window and set it to `match_parent`. This will expand the control so that it occupies as much space as the parent control it lives within. You will learn more about the details of designing layouts in Chapter 8. For now, your layout should resemble Figure 5-12.

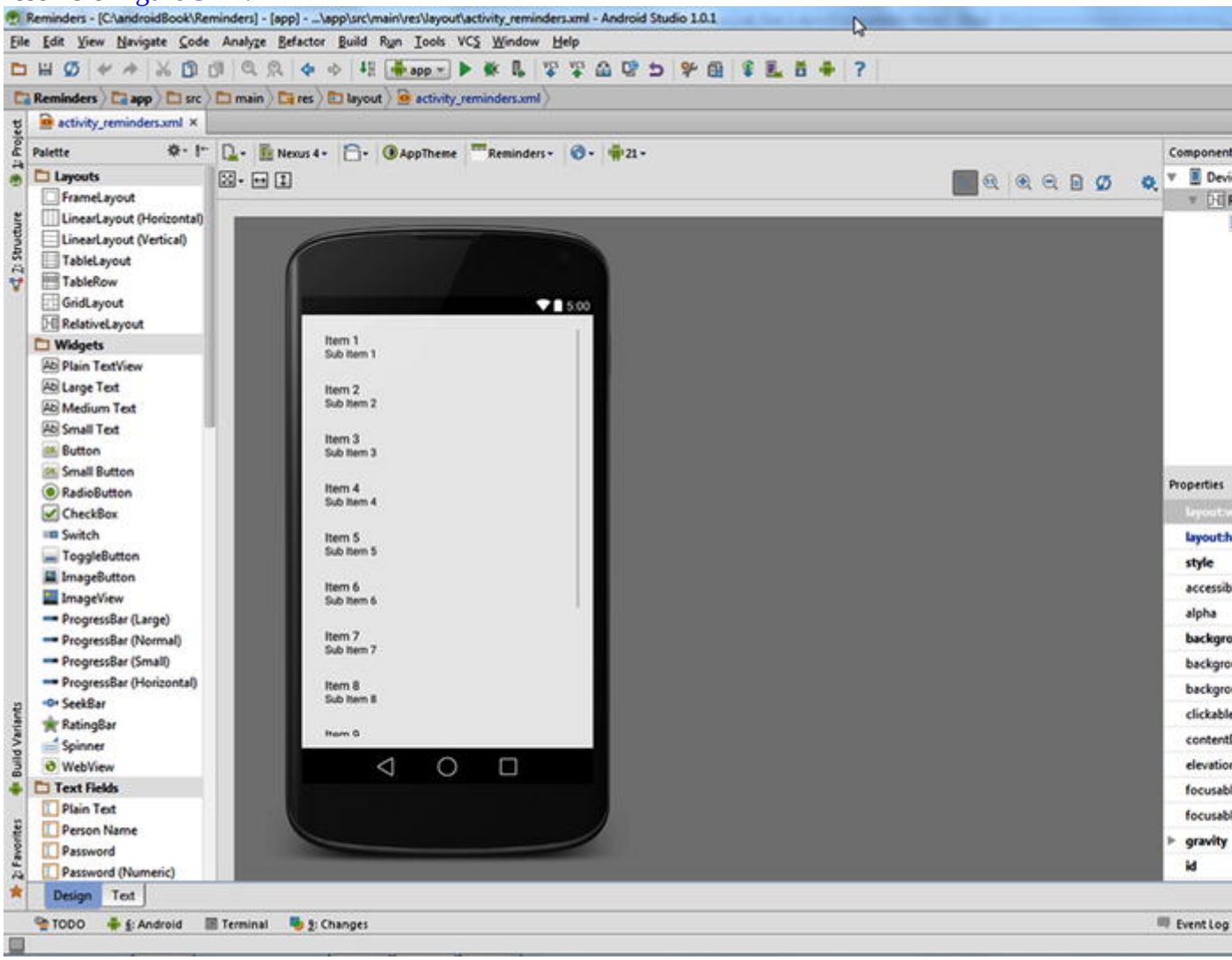


***Figure 5-12.*** *The activity_reminders layout with a ListView*

In Android, an activity defines the logic that controls user interaction with your app. When learning Android for the first time, it helps to think of an activity as a screen within your app, though activities can be more complicated than that. These activities typically inflate a layout which define where things appear on-screen. The layout files are defined as XML but can be edited visually using the Visual Designer as described earlier.

# Editing the Layout's Raw XML

Click the Text tab along the bottom to switch from visual editing to text editing. This brings up a view of the raw XML for the layout, along with a live preview to the right. Changes you make to the XML are immediately reflected in the preview pane. Change the background color of the RelativeLayout to a dark grey by inserting `android:background="#181818"` underneath the line which reads `android:layout_height="match_parent"`. Colors are expressed in hexadecimal values. See Chapter 9 for more information on hexadecimal color values. Notice that there is now a dark-grey swatch that appears in gutter next to the line you inserted which set the background color of the root ViewGroup. If you toggle back to Design mode, you will observe that the entire layout is now dark-grey.

Hard-coding a color value directly in your XML layout file is not the best approach. A better option is to define a `colors.xml` file under the values resource folder and define your colors there. The reason we externalize values to XML files such as colors.xml is that these resources are kept and edited in one place and they can be referenced easily throughout your project.

Select the hex value `#181818` and cut it to your clipboard by using Ctrl+X | Cmd+X or by choosing Edit ➤Cut. Type **@color/dark_grey** in its place. This value uses special syntax to refer to an Android color value named `dark_grey`. This value should be defined in an Android resource file called `colors.xml`, but because this file does not yet exist in your project, Android Studio highlights this error in red. Press Alt+Enter and you will be prompted with options to correct the error. Select the second option, Create Color Value Resource `dark_grey`, and then paste the value in the Resource value: field of the next dialog box that appears and click Ok.

The New Color Value Resource dialog box will create the Android resource file `colors.xml` and fill it with the hexadecimal value. Click OK and then click OK in the Add Files to Git dialog box to have this new file added to version control and be sure to select the Remember, Don't Ask Again checkbox so that you're not bothered with this message again. Figure 5-13 demonstrates this flow.

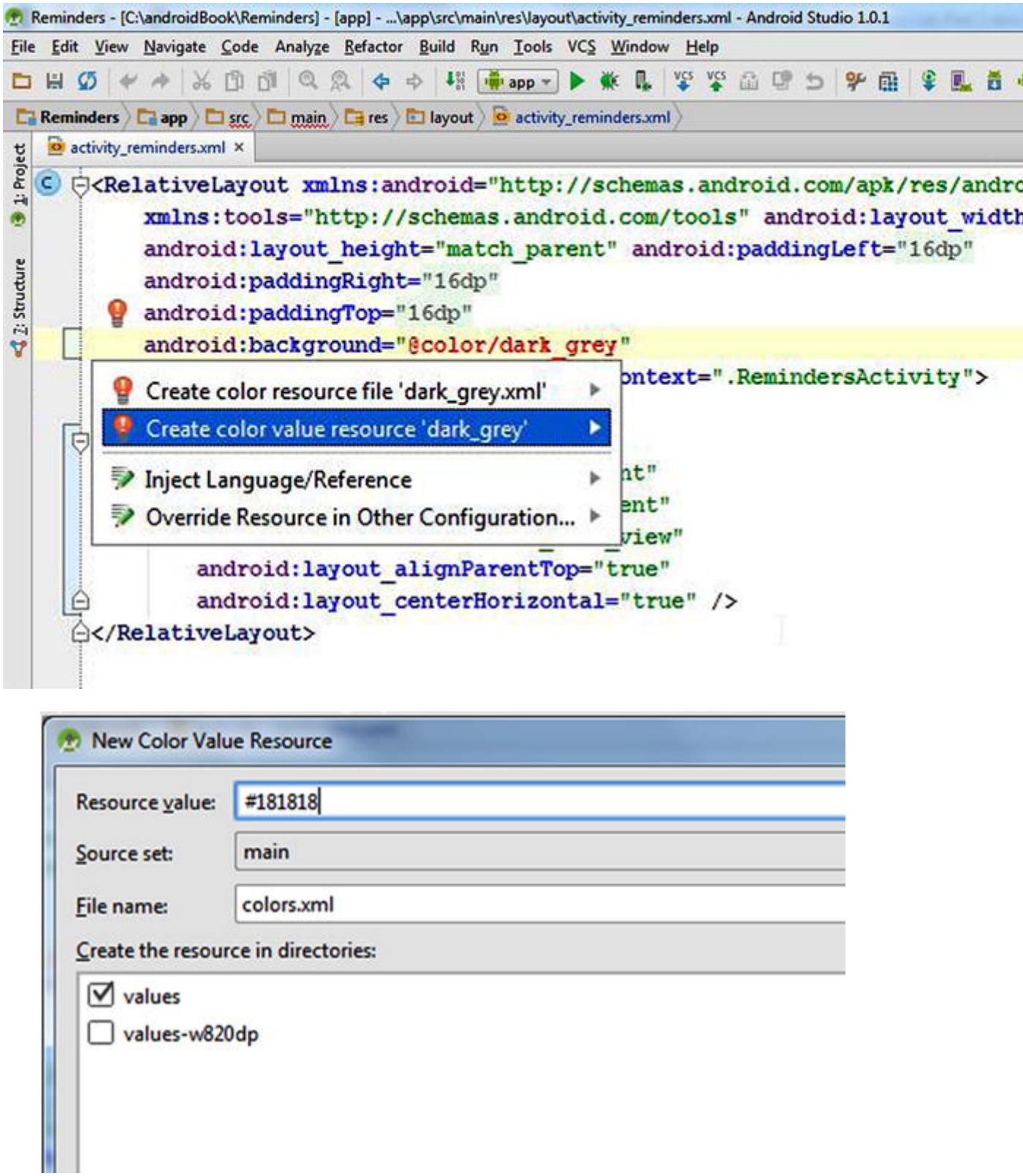


***Figure 5-13.*** *Extracting the hard-coded color value as a resource value*

The `ListView` in preview mode contains row layouts that do not provide enough contrast with our chosen background color. To change the way these items appear, you will define a layout for the row in separate layout file. Right-click the `layout` folder under the `res`

folder and choose New ➤Layout Resource File. Enter **reminders_row** in the New Resource File dialog box. Use `LinearLayout` as the root ViewGroup and keep the rest of the defaults as seen in Figure 5-14.

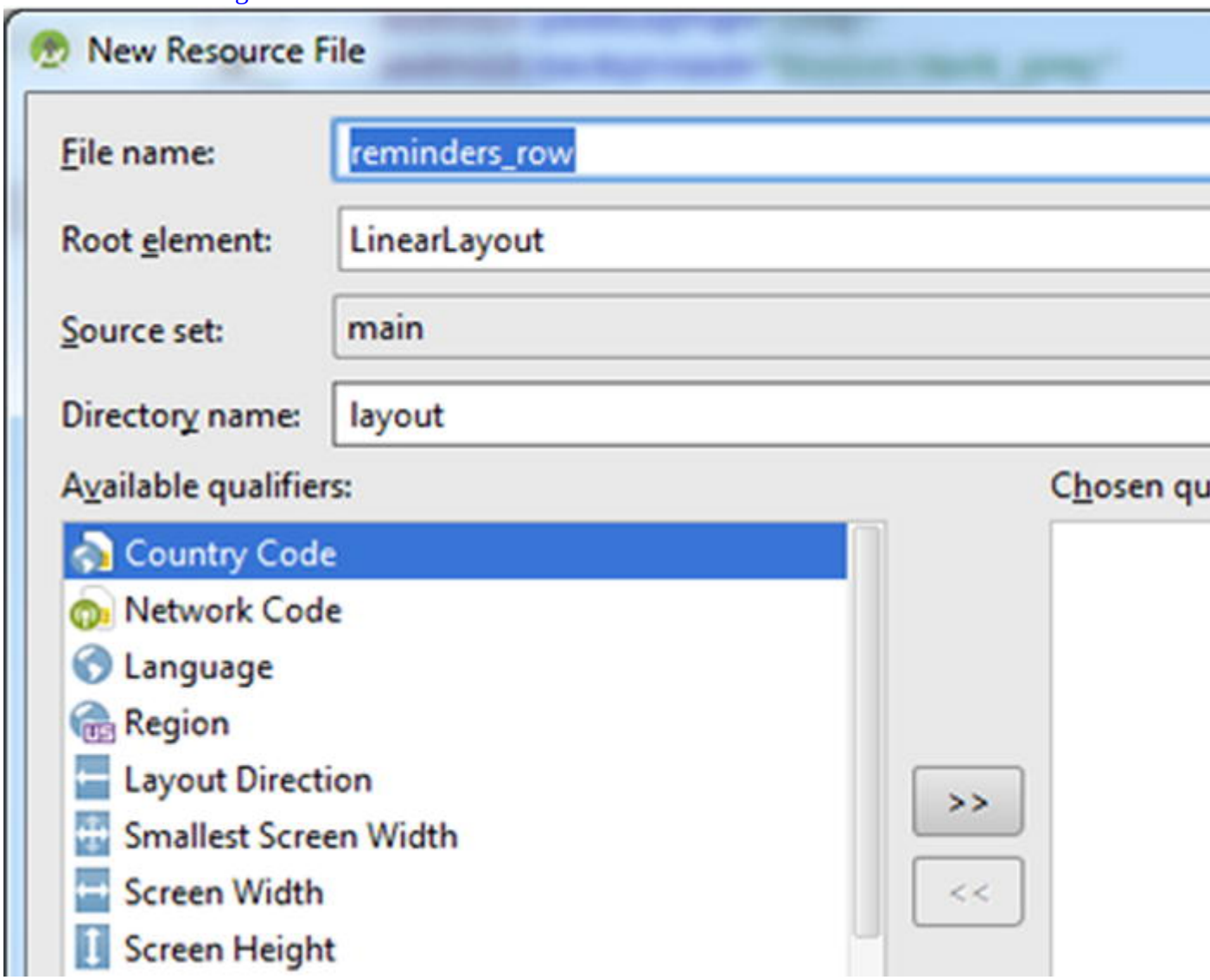


***Figure 5-14.*** *New Resource File dialog box*

You will now create the layout for an individual list item row. The `LinearLayout` root ViewGroup is the outermost element in the layout. Set its orientation to `vertical` by using the controls in the toolbar at the top of the preview pane. Be careful when you use this control because horizontal lines indicate a vertical orientation and vice versa. Figure 5-15 highlights the Change Orientation button.

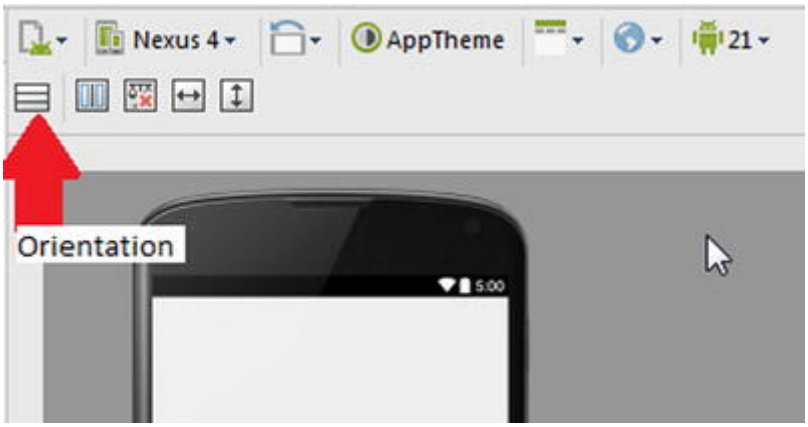


***Figure 5-15.*** *Change Orientation button*

Find the properties view along the bottom right of the preview pane. Find the `layout:height` property and set it to `50dp`. This property controls the height of a control, and the *dp* suffix refers to the *density-independent pixels* measurement. This is a metric that Android uses to allow layouts to scale properly regardless of the screen density on which they are rendered. You can click any property in this view and start typing to incrementally search for properties, and then press the up- or down-arrows to continue searching.

Drag and drop a horizontal `LinearLayout` inside the vertical `LinearLayout`. Drag and drop a `CustomView` control inside the horizontal `LinearLayout` and set its class property to `android.view.View` to create a generic empty view and give it an `id` property of `row_tab`. As of this writing, there is a limitation in Android Studio that does not allow you to drag a generic View from the palette. Once you click `CustomView`, you will get a dialog box with different choices, none of which include the generic View class. Select any class from the dialog and place it in your layout. Find the class property of the view you just placed using the properties window in the properties pane to the right and change it to `android.view.View` to work around this limitation. Refer to Listing 5-1 to see how this is done.

You will use this generic View tab to flag certain reminders as important. With the edit mode still set to Text, change your custom View's `layout:width` property to `10dp` and set its `layout:height` property to `match_parent`. Using the `match_parent` value here will make this `View` control as tall as its parent container. Switch to Design mode and drag and drop a `Large Text` control inside the horizontal `LinearLayout` of the Component Tree and set its `width` and `height` properties to `match_parent`. Verify that your Large Text component is positioned to the right of the custom view control. In the Component Tree, the component labelled textView should be nested inside the LinearLayout (horizontal) component and underneath the view component. If textView appears above view, drag it down with your mouse so that it snaps to the second (and last) position. Give your TextView control an `id` value of `row_text` and set its `textSize` property to `18sp`. The *sp* suffix refers to the *scale-independent pixels* measurement which performs like `dp`, but also respects the user's text size settings so that, for example, if the user were hard of sight and wanted text on her phone to display large, `sp` would respect this setting, whereas `dp` would not. Therefore, it's always a good idea to use `sp` for `textSize`. You will learn more about screen measurements in Chapter 8.

Finally, set the TextView control's `text` property to `Reminder Text`. Switch to Text mode and make additional changes to the XML so that your code resembles Listing 5-1.

***Listing 5-1.*** *The reminders_row Layout XML Code*

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="50dp"
    android:orientation="vertical">

    <LinearLayout
        android:orientation="horizontal"
        android:layout_width="match_parent"
        android:layout_height="48dp">

        <view
            android:layout_width="10dp"
            android:layout_height="match_parent"
            class="android.view.View"
            android:id="@+id/row_tab" />

        <TextView
            android:layout_width="match_parent"
             android:layout_height="50dp"
            android:textAppearance="?
            android:attr/textAppearanceLarge"
            android:text="Reminder Text"
            android:id="@+id/row_text"
            android:textSize="18sp" />
    </LinearLayout>
</LinearLayout>
```

You will now create some custom colors. Switch to Design mode. Select the root LinearLayout (vertical) in the Component Tree. Set its android:background attribute to `@color/dark_grey` to reuse the color we defined earlier. Select the row_tab component in the Component Tree and set its `android:background` attribute to `@color/green`. Select the row_text component and set its `android:textColor` attribute to `@color/white`. As before, these colors are not defined in the `colors.xml` file, and you will need to use the same process as before to define them. Switch to Text mode. Press the F2 key repeatedly to jump back and forth between these two additional errors and press Alt+Enter to bring up the IntelliSense suggestion. Choose the second suggestion in both cases and fill in the pop-up dialog box with the values `#ffffff` for fixing the white color and `#003300` to fix the green color. After using the suggestion dialog box to fix these errors, you can hold the Ctrl key and left-click any of these colors which will bring you to the `colors.xml` file and should look like Listing 5-2.

***Listing 5-2.*** *The colors.xml File*

```
<resources>
    <color name="dark_grey">#181818</color>
    <color name="white">#ffffff</color>
    <color name="green">#003300</color>
</resources>
```

Return to the `activity_reminders.xml` layout file. You will now connect the new `reminders_row` layout to the `ListView` in this layout. Switch to Text mode and add the following attribute to the `ListView` element:
`tools:listitem="@layout/reminders_row"` as shown in Figure 5-16.

Adding this attribute doesn't change the way the layout renders when it runs; it merely changes what the preview pane uses for each item in the list view. To make use of the new layout, you must inflate it using Java code and we will show you how do that in a subsequent step.

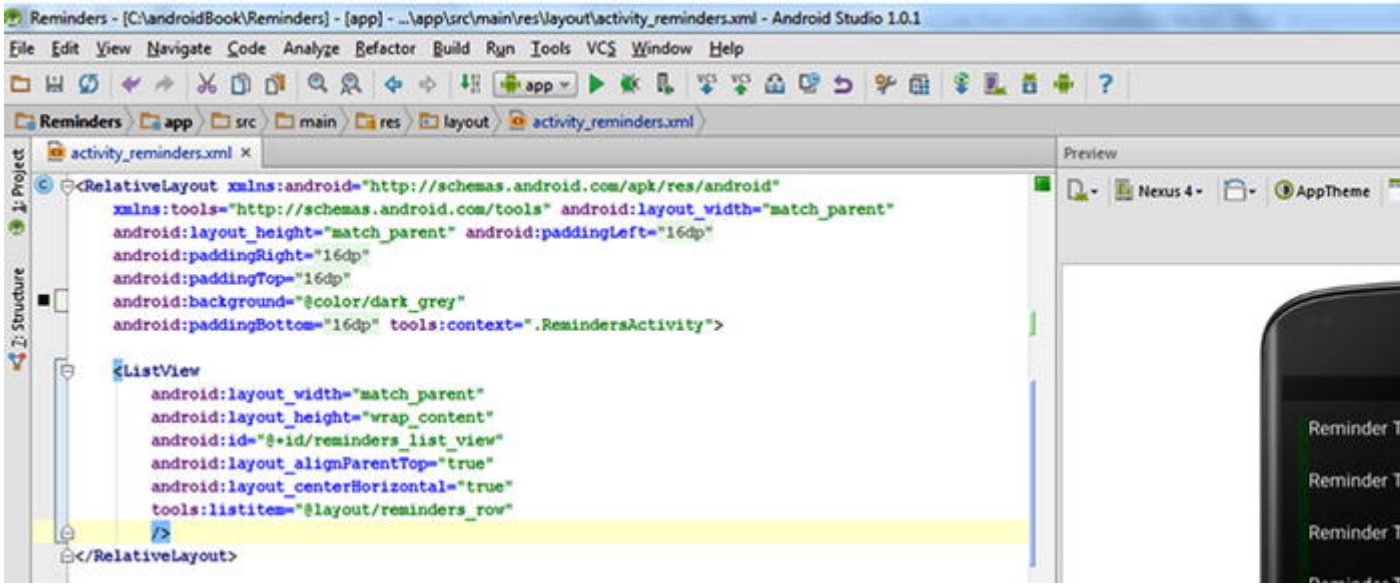


***Figure 5-16.*** *The preview pane is now rendering a custom ListView layout*

# Adding Visual Enhancements

You have just completed a custom layout for your ListView rows, but you shouldn't stop there. Adding a few visual enhancements will make your app stand-out from the others. Take a look at how the text renders on-screen. A careful eye will catch how it is slightly off-center and runs up against the green tab on the left. Open the `reminders_row` layout to make some minor adjustments. You want the text to gravitate toward the vertical center of the row and give a bit of padding so as to provide some visual separation from the side edges. Replace your `TextView` element with the code in Listing 5-3.

***Listing 5-3.*** *TextView Additional Attributes*

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="50dp"
    android:text="Reminder Text"
    android:id="@+id/row_text"
    android:textColor="@color/white"
    android:textSize="18sp"
    android:gravity="center_vertical"
    android:padding="10dp"
    android:ellipsize="end"
    android:maxLines="1"
    />
```

The additional `ellipsize` attribute will truncate text that is too long to fit in the row with an ellipsis on the end, whereas the `maxLines` attribute restricts the number of lines in each row to 1. Finally, add two more generic view objects from Listing 5-4 after the inner `LinearLayout` but before the closing tag of the outer `LinearLayout` to create a horizontal rule beneath the row. The outer `LinearLayout` is set to a height of `50dp`, and the inner `LinearLayout` is set to a height of `48dp`. The two generic view objects will

occupy the remaining vertical 2dp inside the layout creating a beveled edge. This is shown in Listing 5-4.

***Listing 5-4.*** *Extra Generic Views for beveled edge*

```
</LinearLayout>
<view
    class="android.view.View"
    android:layout_width="fill_parent"
    android:layout_height="1dp"
    android:background="#000"/>
<view
    class="android.view.View"
    android:layout_width="fill_parent"
    android:layout_height="1dp"
    android:background="#333"/>
</LinearLayout>
```

# Adding Items to ListView

You will now make changes to the activity that uses the layout you just modified. Open the Project tool window and find the `RemindersActivity` file under your `java` source folder. It will be located under the `com.apress.gerber.reminders` package. Find the `onCreate()` method in this file. It should be the first method defined in your class. Declare a ListView member called mListView and change the `onCreate()` method to look like the code in Listing 5-5. You will need to resolve the imports ListView and ArrayAdapter.

***Listing 5-5.*** *Add List Items to the ListView*

```
public class RemindersActivity extends ActionBarActivity {

    private ListView mListView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_reminders);
        mListView = (ListView) findViewById(R.id.reminders_list_view);
        //The arrayAdatper is the controller in our
        //model-view-controller relationship. (controller)
        ArrayAdapter<String> arrayAdapter = new ArrayAdapter<String>(
                //context
                this,
                //layout (view)
                R.layout.reminders_row,
                //row (view)
                R.id.row_text,
                //data (model) with bogus data to test our listview
                new String[]{"first record", "second record", "third
record"});

        mListView.setAdapter(arrayAdapter);
    }
    //Remainder of the class listing omitted for brevity
}
```

This code looks up the `ListView` by using the `id` property you defined earlier and removes the the default list divider so that the custom beveled divider we created earlier will render properly. The code also creates an adapter with a few example list items. `Adapter` is a special Java class defined as part of the Android SDK that functions as the Controller in the Model-View-Controller relationship among the SQLite database (Model), the ListView

(View), and the Adapter (Controller). The Adapter binds the Model to the View and handles updates and refreshes. Adapter is the superclass of ArrayAdapter, which binds elements of an Array to a View. In our case, this View is a ListView. `ArrayAdapter` takes three parameters in its three-argument constructor. The first parameter is a `Context` object represented by the current activity. The Adapter also needs to know which layout and which field, or fields, in the layout should be used to display row data. To satisfy this requirement, you pass the ids of both the layout and the `TextView` item in the layout. The last parameter is an array of Strings used for each item in the list. If you run the project at this point, you will see the values given in the `ArrayAdapter` constructor displayed in the list view as seen in Figure 5-17.


***Figure 5-17.*** *ListView example*

Press Ctrl+K | Cmd+K to commit your changes to Git and use **Adds ListView with custom colors** as the commit message. As you work through a project, it is good practice to perform incremental commits to Git while using commit messages that describe the features each

commit adds/removes/changes. Keeping this habit makes it easy to identify individual commits and later build release notes for future collaborators and users.

# Setting the Action Bar Overflow Menu

Android uses a common visual element called Action Bar. The Action Bar is where many apps locate navigation and other options that allow the user to perform important tasks. When you run the app at this point, you may notice a menu icon that looks like three vertical dots. These dots are known as the overflow menu. Clicking the overflow menu icon produces a menu with a single menu item entry called `settings`. This menu item is placed there as part of the new project wizard template and is essentially a placeholder that performs no action. The `RemindersActivity` loads the `menu_reminders.xml` file, which is found under the `res/menu` folder. Make changes to this file to add new menu items to the activity as seen in Listing 5-6.

***Listing 5-6.*** *New Menu Items*

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context="com.apress.gerber.reminders.app.RemindersActivity" >
    <item android:id="@+id/action_new"
        android:title="new Reminder"
        android:orderInCategory="100"
        app:showAsAction="never" />
    <item android:id="@+id/action_exit"
        android:title="exit"
        android:orderInCategory="200"
        app:showAsAction="never" />
</menu>
```

In the preceding code listing, the `title` attribute corresponds to the text displayed in the menu item. Since we've hard-coded these attributes, Android Studio will flag these values as warnings. Press F2 to jump between these warnings and press Alt+Enter to pull up the IntelliSense suggestions. You simply need to press Enter to accept the first suggestion, type a name for the new String resource, and as soon as the dialog box pops-up, press Enter again to accept the named resource. Use `new_reminder` for the name of the first item and `exit` for the second.

Open `RemindersActivity` and replace the `onOptionsItemSelected()` method with the text in Listing 5-7. You will need to resolve the import for the Log class. When you tap a menu item in the app, the runtime invokes this method, passing in a reference to whichever `MenuItem` was tapped. The `switch` statement takes the `itemId` of the `MenuItem` and either performs a log statement or terminates the activity, depending on which item was tapped. This example uses the `Log.d()` method that writes text to the Android debug logs. If your app contained multiple activities and those activities were viewed prior to the current activity, then calling `finish()` would simply pop the current activity off the backstack and control would pass to the next underlying activity. Because the RemindersActivity is the only activity in this app, the `finish()` method pops the one and only activity off the backstack and results in the termination of your app.

***Listing 5-7.*** *onOptionsItemSelected() Method Definition*

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action_new:
            //create new Reminder
```

```
            Log.d(getLocalClassName(),"create new Reminder");
            return true;
        case R.id.action_exit:
            finish();
            return true;
        default:
            return false;
    }
}
```

Run the app and test the new menu options. Tap the new Reminder menu option and watch the Android log to see the message appear. The Android DDMS (Dalvik Debug Monitor Service) window will open as you run your app on your emulator or device, and you will need to select the Debug option under Log Level to see debug logs. Run your app and interact with the menu items. Pay attention to the logs in the Android DDMS window as you tap the New Reminder menu item. Finally, press Ctrl+K | Cmd+K and commit your code to Git using **Adds new reminder and exit menu options** as your commit message.

# Persisting Reminders

Because the Reminders app will need to maintain a list of reminders, you will need a persistence strategy. The Android SDK and runtime provide an embedded database engine called *SQLite*, which is designed to operate in constrained memory environments and is well suited for mobile devices. This section covers the SQLite database and explores how to maintain a list of reminders. Our strategy will include a data model, a database proxy class, and a `CursorAdapter`. The model will hold the data that is read from and written to the database. The proxy will be an adapter class that will translate simple calls from the app into API calls to the SQLite database. Finally, the `CursorAdapter` will extend a standard Android class that deals with data access in an abstract way.

## Data Model

Let's start by creating the data model. Right click the com.apress.gerber.reminders package and select New ➤Java Class. Name your class Reminder and press Enter. Decorate your class with the code in Listing 5-8. This class is a simple POJO (Plain Old Java Object) that defines a few instance variables and corresponding getter and setter methods. The `Reminder` class includes an integer ID, a String value, and a numeric importance value. The ID is a unique number used to identify each reminder. The String value holds the text for the reminder. The importance value is a numeric indicator that flags an individual reminder as important (1 = important, 0 = not important). We used `int` rather than `boolean` here because the SQLite database does not have a `boolean` datatype.

***Listing 5-8.*** *Reminder Class Definition*

```
public class Reminder {

    private int mId;
    private String mContent;
    private int mImportant;

    public Reminder(int id, String content, int important) {
        mId = id;
        mImportant = important;
        mContent = content;
    }
```

```
    public int getId() {
        return mId;
    }

    public void setId(int id) {
        mId = id;
    }

    public int getImportant() {
        return mImportant;
    }

    public void setImportant(int important) {
        mImportant = important;
    }

    public String getContent() {
        return mContent;
    }

    public void setContent(String content) {
        mContent = content;
    }
}
```

Now you will create a proxy to the database. Again, this proxy will translate simple application calls into lower-level SQLite API calls. Create a new class in your com.apress.gerber.reminders package called `RemindersDbAdapter`. Place the code in Listing 5-9 directly inside of your newly created RemindersDbAdapter class. As you resolve imports, you will notice that DatabaseHelper is not found in the Android SDK. We will define the DatabaseHelper class in a subsequent step. This code defines the column names and indices; a `TAG` for logging; two database API objects; some constants for the database name, version, and the main table name; the context object; and a SQL statement used to create the database.

***Listing 5-9.*** *Code to be placed inside the RemindersDbAdapter class*

```
//these are the column names
public static final String COL_ID = "_id";
public static final String COL_CONTENT = "content";
public static final String COL_IMPORTANT = "important";

//these are the corresponding indices
public static final int INDEX_ID = 0;
public static final int INDEX_CONTENT = INDEX_ID + 1;
public static final int INDEX_IMPORTANT = INDEX_ID + 2;

//used for logging
private static final String TAG = "RemindersDbAdapter";

private DatabaseHelper mDbHelper;
private SQLiteDatabase mDb;

private static final String DATABASE_NAME = "dba_remdrs";
private static final String TABLE_NAME = "tbl_remdrs";
private static final int DATABASE_VERSION = 1;

private final Context mCtx;

//SQL statement used to create the database
private static final String DATABASE_CREATE =
```

```
        "CREATE TABLE if not exists " + TABLE_NAME + " ( " +
                COL_ID + " INTEGER PRIMARY KEY autoincrement, " +
                COL_CONTENT + " TEXT, " +
                COL_IMPORTANT + " INTEGER );";
```

# SQLite API

`DatabaseHelper` is a SQLite API class used to open and close the database. It uses `Context`, which is an abstract Android class that provides access to the Android operating system. `DatabaseHelper` is a custom class, and must be defined by you. Use the code in Listing 5-10 to implement `DatabaseHelper` as an inner class of `RemindersDbAdapter`. Place this proceeding code towards the end of the RemindersDbAdatper but still inside RemindersDbAdapters enclosing braces.

***Listing 5-10.*** *RemindersDbAdapter*

```
private static class DatabaseHelper extends SQLiteOpenHelper {
    DatabaseHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        Log.w(TAG, DATABASE_CREATE);
        db.execSQL(DATABASE_CREATE);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int
newVersion) {
        Log.w(TAG, "Upgrading database from version " + oldVersion + " to "
                + newVersion + ", which will destroy all old data");
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
        onCreate(db);
    }
}
```

`DatabaseHelper` extends `SQLiteOpenHelper`, which helps maintain the database with special callback methods. Callback methods are methods that the runtime environment will call throughout the life-cycle of the application, and they use the `SQLiteDatabase db` variable supplied to execute SQL commands. The constructor is where the database is initialized. The constructor passes the database name and version to its superclass; and then the superclass does the hard work of setting up the database. The `onCreate()` method is called automatically by the runtime when it needs to create the database. This operation runs only once, when the app first launches and the database has not yet been created. The `onUpgrade()` method is called whenever the database needs to be upgraded, for example if the developer changes the schema. If you do change the database schema, be sure to increment the `DATABASE_VERSION` by one, and `onUpgrade()` will manage the rest. If you forget to increment the `DATABASE_VERSION`, your app will crash even in debug build mode. In the preceding code, we run a SQL command to drop the one and only table in the database before running the `onCreate()` method to re-create the table.

The code in Listing 5-11 demonstrates using `DatabaseHelper` to open and close the database. The constructor saves an instance of `Context`, which is passed to `DatabaseHelper`. The `open()` method initializes the helper and uses it to get an instance of the database, while the `close()` method uses the helper to close the database. Add this

code after all the member variable definitions and before the `DatabaseHelper` inner class definition inside the RemindersDbAdapter class. When you resolve imports, use the `android.database.SQLException` class.

***Listing 5-11.*** *Database Open and Close Methods*

```
public RemindersDbAdapter(Context ctx) {
    this.mCtx = ctx;
}

//open
public void open() throws SQLException {
    mDbHelper = new DatabaseHelper(mCtx);
    mDb = mDbHelper.getWritableDatabase();
}

//close
public void close() {
    if (mDbHelper != null) {
        mDbHelper.close();
    }
}
```

Listing 5-12 contains all the logic that handles the creating, reading, updating, and deleting of `Reminder` objects in the `tbl_remdrs` table. These are usually referred to as *CRUD* operations; CRUD stands for create, read, update, delete. Add the proceeding code after the `close()` method inside the RemindersDbAdapter class.

***Listing 5-12.*** *Database CRUD Operations*

```
//CREATE
//note that the id will be created for you automatically
public void createReminder(String name, boolean important) {
    ContentValues values = new ContentValues();
    values.put(COL_CONTENT, name);
    values.put(COL_IMPORTANT, important ? 1 : 0);
    mDb.insert(TABLE_NAME, null, values);
}

//overloaded to take a reminder
public long createReminder(Reminder reminder) {
    ContentValues values = new ContentValues();
    values.put(COL_CONTENT, reminder.getContent()); // Contact Name
    values.put(COL_IMPORTANT, reminder.getImportant()); // Contact Phone
Number

    // Inserting Row
    return mDb.insert(TABLE_NAME, null, values);
}

//READ
public Reminder fetchReminderById(int id) {
    Cursor cursor = mDb.query(TABLE_NAME, new String[]{COL_ID,
                    COL_CONTENT, COL_IMPORTANT}, COL_ID + "=?",
            new String[]{String.valueOf(id)}, null, null, null, null
    );
    if (cursor != null)
        cursor.moveToFirst();

    return new Reminder(
            cursor.getInt(INDEX_ID),
            cursor.getString(INDEX_CONTENT),
            cursor.getInt(INDEX_IMPORTANT)
```

```
    );
}

public Cursor fetchAllReminders() {
    Cursor mCursor = mDb.query(TABLE_NAME, new String[]{COL_ID,
                    COL_CONTENT, COL_IMPORTANT},
            null, null, null, null, null
    );

    if (mCursor != null) {
        mCursor.moveToFirst();
    }
    return mCursor;
}

//UPDATE
public void updateReminder(Reminder reminder) {
    ContentValues values = new ContentValues();
    values.put(COL_CONTENT, reminder.getContent());
    values.put(COL_IMPORTANT, reminder.getImportant());
    mDb.update(TABLE_NAME, values,
            COL_ID + "=?", new String[]{String.valueOf(reminder.getId())});
}

//DELETE
public void deleteReminderById(int nId) {
    mDb.delete(TABLE_NAME, COL_ID + "=?", new
String[]{String.valueOf(nId)});
}

public void deleteAllReminders() {
    mDb.delete(TABLE_NAME, null, null);
}
```

Each of these methods uses the `SQLiteDatabase mDb` variable to generate and execute SQL statements. If you are familiar with SQL, you may guess that these SQL statements will be in the form of an `INSERT`, `SELECT`, `UPDATE`, or `DELETE`.

The two create methods use a special `ContentValues` object, which is a data shuttle used to pass data values to the database object's `insert` method. The database will eventually convert these objects into SQL `insert` statements and execute them. There are two `read` methods, one for fetching a single reminder and another for fetching a cursor to iterate all reminders. You will use `Cursor` later in a special `Adapter` class.

The `update` method is similar to the second create method. However, this method calls an `update` method on the lower-level database object, which will generate and execute an `update` SQL statement rather than an `insert`.

Last, there are two `delete` methods. The first takes an `id` parameter and uses the database object to generate and execute a `delete` statement for a particular reminder. The second method requests that the database generate and execute a `delete` statement to remove all the reminders from the table.

At this point, you need a means of getting reminders out of the database and into the `ListView`. Listing 5-13 demonstrates the logic necessary to bind database values to individual row objects by extending the special `Adapter` Android class you saw earlier. Create a new class called RemindersSimpleCursorAdapter in the

com.apress.gerber.reminders package and decorate it with the proceeding code. As you resolve imports, use the android.support.v4.widget.SimpleCursorAdapter class.

***Listing 5-13.*** *RemindersSimpleCursorAdapter Code*

```
public class RemindersSimpleCursorAdapter extends SimpleCursorAdapter {
    public RemindersSimpleCursorAdapter(Context context, int layout, Cursor
c, String[] from, int[] to, int flags) {
        super(context, layout, c, from, to, flags);
    }

    //to use a viewholder, you must override the following two methods and
define a ViewHolder class
    @Override
    public View newView(Context context, Cursor cursor, ViewGroup parent) {
        return super.newView(context, cursor, parent);
    }

    @Override
    public void bindView(View view, Context context, Cursor cursor) {
        super.bindView(view, context, cursor);

        ViewHolder holder = (ViewHolder) view.getTag();
        if (holder == null) {
            holder = new ViewHolder();
            holder.colImp
= cursor.getColumnIndexOrThrow(RemindersDbAdapter.COL_IMPORTANT);
            holder.listTab =  view.findViewById(R.id.row_tab);
            view.setTag(holder);
        }

        if (cursor.getInt(holder.colImp) > 0) {
            holder.listTab.setBackgroundColor(context.getResources().getCol
or(R.color.orange));
        } else {
            holder.listTab.setBackgroundColor(context.getResources().getCol
or(R.color.green));
        }
    }

    static class ViewHolder {
        //store the column index
        int colImp;
        //store the view
        View listTab;
    }
}
```

We register the `Adapter` with the `ListView` to populate reminders. During runtime, the `ListView` will repeatedly invoke the `bindView()` method on the `Adapter` with individual onscreen `View` objects as the user loads and scrolls through the list. It is the job of the `Adapter` to fill these views with list items. In this code example, we're using a subclass of `Adapter` called `SimpleCursorAdapter`. This class uses a `Cursor` object, which keeps track of the rows in the table.

Here you see an example of the `ViewHolder` pattern. This is a well-known Android pattern in which a small `ViewHolder` object is attached as a tag on each view. This object adds decoration for `View` objects in the list by using values from the data source, which in this example is the `Cursor`. The `ViewHolder` is defined as a static inner class with two

instance variables, one for the index of the Important table column and one for the `row_tab` view you defined in the layout.

The `bindView()` method starts by calling the superclass method that maps values from the cursor to elements in the View. It then checks to see whether a holder has been attached to the tag and creates a new holder if necessary. The `bindView()` method then configures the holder's instance variables by using both the Important column index and the `row_tab` you defined earlier. After the holder is either found or configured, it uses the value of the `COL_IMPORTANT` constant from the current reminder to decide which color to use for the `row_tab`. The example uses a new orange color, which you need to add to your `colors.xml: <color name="orange">#ffff381a</color>`.

Earlier you used an `ArrayAdapter` to manage the relationship between model and view. The `SimpleCursorAdapter` follows the same pattern, though its model is an SQLite database. Make the changes in Listing 5-14 to use your new RemindersDbAdapter and RemindersSimpleCursorAdapter.

***Listing 5-14.*** *RemindersActivity Code*

```
public class RemindersActivity extends ActionBarActivity {
    private ListView mListView;
    private RemindersDbAdapter mDbAdapter;
    private RemindersSimpleCursorAdapter mCursorAdapter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_reminders);
        mListView = (ListView) findViewById(R.id.reminders_list_view);
        mListView.setDivider(null);
        mDbAdapter = new RemindersDbAdapter(this);
        mDbAdapter.open();

        Cursor cursor = mDbAdapter.fetchAllReminders();

        //from columns defined in the db
        String[] from = new String[]{
                RemindersDbAdapter.COL_CONTENT
        };

        //to the ids of views in the layout
        int[] to = new int[]{
                R.id.row_text
        };

        mCursorAdapter = new RemindersSimpleCursorAdapter(
                //context
                RemindersActivity.this,
                //the layout of the row
                R.layout.reminders_row,
                //cursor
                cursor,
                //from columns defined in the db
                from,
                //to the ids of views in the layout
                to,
                //flag - not used
                0);

        //the cursorAdapter (controller) is now updating the listView
```

```
(view)
        //with data from the db (model)
        mListView.setAdapter(mCursorAdapter);
    }
//Abbreviated for brevity
}
```

If you run the app at this point, you will not see anything in the list; the screen will be completely empty because your last change inserted the SQLite functionality in place of the example data. Press Ctrl+K | Cmd+K and commit your changes with the message **Adds SQLite database persistence for reminders and a new color for important reminders**. As a challenge, you might try to figure out how to add the example items back by using the new `RemindersDbAdapter`. This is covered in the next chapter, so you can look ahead and check your work.

# Summary

At this point, you have a maturing Android app. In this chapter, you learned how to set-up your first Android project and controlled its source using Git. You also explored how to edit Android layouts in both Design and Text mode. You have seen a demonstration of creating an overflow menu in the Action Bar. The chapter concluded by exploring `ListViews` and `Adapters`, and binding data to the built-in SQLite database. In the following chapter, you will complete the app by adding the ability to create and edit reminders.