

Android Development 1

Lesson 1: **Getting Started with Android Development**

[About Eclipse](#)

[Perspectives and the Red Leaf Icon](#)

[Working Sets](#)

[Hello, Android!](#)

[Create the Project](#)

[Run the Application](#)

[Editing Programs](#)

[Android Package Structure](#)

[Bonus Round](#)

[Wrapping Up](#)

Lesson 2: **Activities and Views**

[AndroidManifest.xml](#)

[Activity Class](#)

[Basic View Components: Layouts and Buttons](#)

[Layouts](#)

[View Components](#)

[Wrapping Up](#)

Lesson 3: **Navigation with Data**

[Working with Intent](#)

[An Emulator Email Alternative](#)

[Sharing Data Between Activities](#)

[Sending Data to a New Activity](#)

[Returning Data to the Previous Activity](#)

[Application Class](#)

[Wrapping Up](#)

Lesson 4: **Android Resources**

[String Resources](#)

[Loading Strings in XML](#)

[Loading Strings in Code](#)

[The Resource Values Folder](#)

[Wrapping Up](#)

Lesson 5: **Drawables - Image Basics**

[Drawable Folders and Qualifiers](#)

[Using Drawables](#)

[Dimensions](#)

[Image Padding](#)

[The ImageButton Widget](#)

[Wrapping Up](#)

Lesson 6: **Lists**

Implementing an Android List

ListView

ListActivity

Empty Lists

ListAdapter

Sorting the Adapter

Overriding ArrayAdapter

List Interaction

Wrapping Up

Lesson 7: **Dialogs, New and Old**

Old Style

AlertDialog

Custom Dialog

New Style

Support Library

Fragments

DialogFragment

Wrapping Up

Lesson 8: **Menus**

Menus, Menus, Menus

Options Menu

Modifying an Options Menu

Context Menu

Wrapping Up

Lesson 9: **Saving Data with Shared Preferences**

Shared Preferences

Getting Started with SharedPreferences

PreferenceActivity

Wrapping Up

Lesson 10: **Saving Data with a Database**

SQLite

Creating a Helper

Using the Helper

Cursor and CursorAdapter

Wrapping Up

Lesson 11: **Threading with AsyncTasks**

Threading in Android

AsyncTask

Tracking Progress

Wrapping Up

Lesson 12: **Styles and Themes**

Introduction to Styling

Defining Styles

[Defining Themes](#)

[Style Inheritance](#)

[Direct Theme References](#)

[Learning to Learn](#)

[Wrapping Up](#)

Lesson 13: **[Android Final Project](#)**

[Final Project](#)

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Getting Started with Android Development

Welcome to the O'Reilly School of Technology **Android 1** course! We're glad you've decided to take this journey with us into Android application development. By the time you finish the course, we're confident that you'll have a firm grasp on developing applications for the Android platform.

Course Objectives

When you complete this course, you will be able to:

- use basic view components and application classes.
- program strings, drawables, and lists.
- display dialogs, menus, styles, and themes.
- save and manipulate data using Shared Preferences and SQLite databases.
- use thread processes.
- create an application that implements multiple activities and can interact with a SQLite database.

In this course, you will learn the fundamentals of writing Android applications. Topics covered include activities, views, navigation with data, drawables, lists, menus, saving data with an SQLite database, and threading. By the end of the course, you will be able to create an application that implements multiple activities and can interact with an SQLite database.

To be successful in this course, you must have a basic understanding of object-oriented programming and the Java programming language. If either of those are unfamiliar to you, talk to your instructor about taking the O'Reilly School of Technology [Object Oriented Java course](#).

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take a *user-active* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.
- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.

- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

CODE TO TYPE:
White boxes like this contain code for you to try out (type into a file to run).
If you have already written some of the code, new code for you to add <code>looks like this</code> .
If we want you to remove existing code, the code to remove <code>will look like this</code> .
We may also include instructive comments that you don't need to type.

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

INTERACTIVE SESSION:
The plain black text that we present in these INTERACTIVE boxes is provided by the system (not for you to type). The commands we want you to type <code>look like this</code> .

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

OBSERVE:
Gray "Observe" boxes like this contain information (usually code specifics) for you to <i>observe</i> .

The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

Note Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

Tip Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

WARNING Warnings provide information that can help prevent program crashes and data loss.

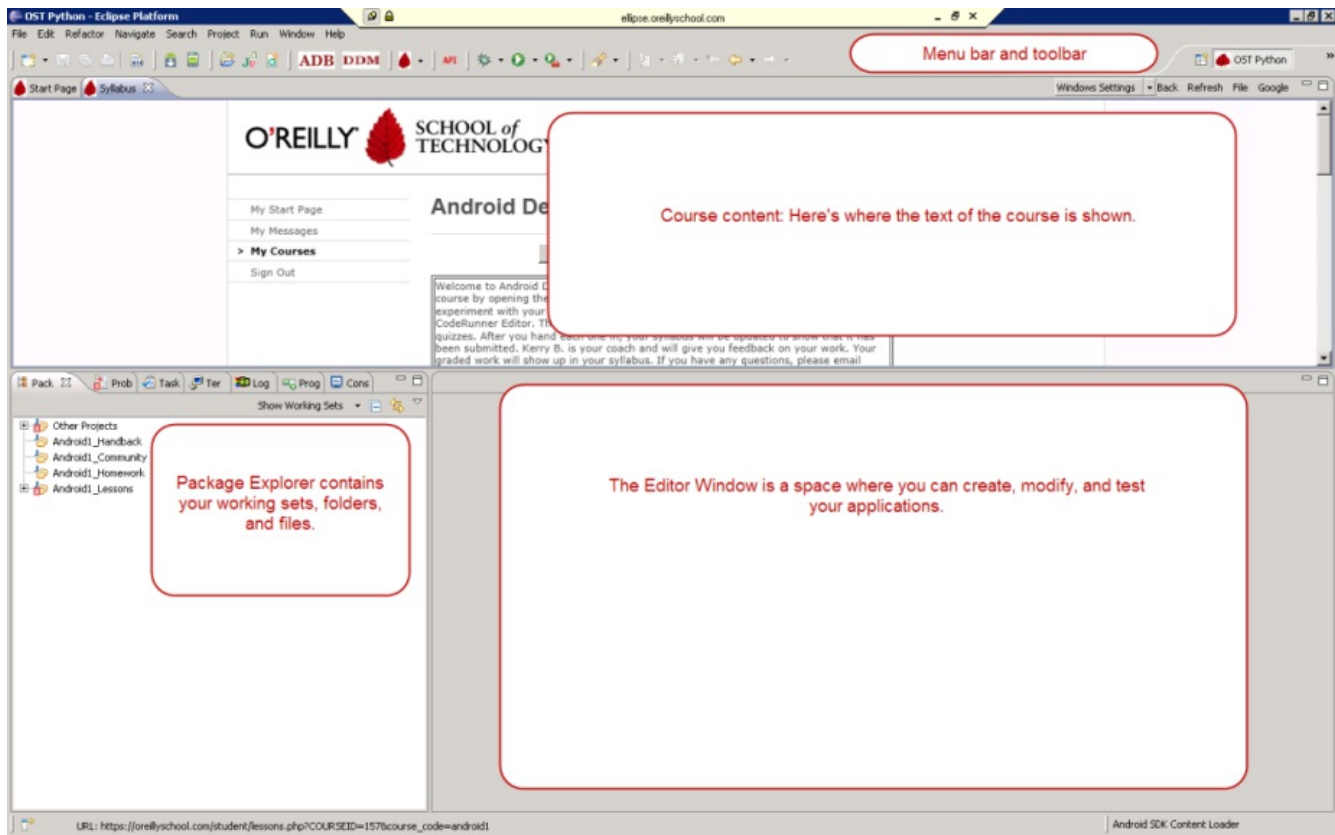
About Eclipse

We're using an Integrated Development Environment (IDE) called Eclipse. It's the program filling up your screen right now. IDEs assist programmers by performing many of the tasks that need to be done repetitively. IDEs can also help to edit and debug code, and organize projects.

Note

You'll make some changes to your working environment during this lesson, so when you complete the lesson, you'll need to exit Eclipse to save those changes.

The Eclipse window displays lesson content, and provides space for you to create, manage, and run programs:



Perspectives and the Red Leaf Icon

The Eclipse Plug-in for Eclipse, developed by the O'Reilly School of Technology, adds an icon to the tool bar in Eclipse. This icon is your "panic button." Since Eclipse is so versatile, you are allowed to move things around, like views, toolbars, and such. If you become confused and want to return to the default perspective (window layout), clicking on the Red Leaf icon allows you to do that right away.



The icon has these functions:

- It allows you to reset the current perspective, by clicking the icon.
- It allows you to change perspectives by clicking the drop-down arrow beside the Red Leaf icon and selecting a series name (ANDROID, JAVA, PYTHON, C++, etc.). Most of the perspectives look similar, but subtle changes may be present "under the hood," so it's best to use the correct perspective for the course. For this course, select **Android**.



Working Sets

All projects created in Eclipse exist in the workspace directory of your account on our server. As you create multiple projects for each lesson in each course, it's possible that your workspace directory could become pretty cluttered. To help alleviate the potential clutter, in this course, we'll use *working sets*. A working set is a logical view of the workspace; it behaves like a folder, but it's really just an association of files. Working sets

allow you to limit the detail that you see at any given time. The difference between a working set and a folder is that a working set doesn't actually exist in the file system. A working set is a convenient way to group related items together. You can assign a project to one or more working sets. In some cases, like with the Android ADT plugin to Eclipse, new projects are created without regard for working sets and will be placed in the workspace, but not assigned to a working set (appearing in the "Other Projects" working set). To assign one of these projects to a working set, right-click on the project name and select the **Assign Working Sets** menu item.

We've created some working sets in the Eclipse IDE for you already. To turn the working set display on and off in Eclipse, see [these instructions](#).

Setting Up Your Android Emulator

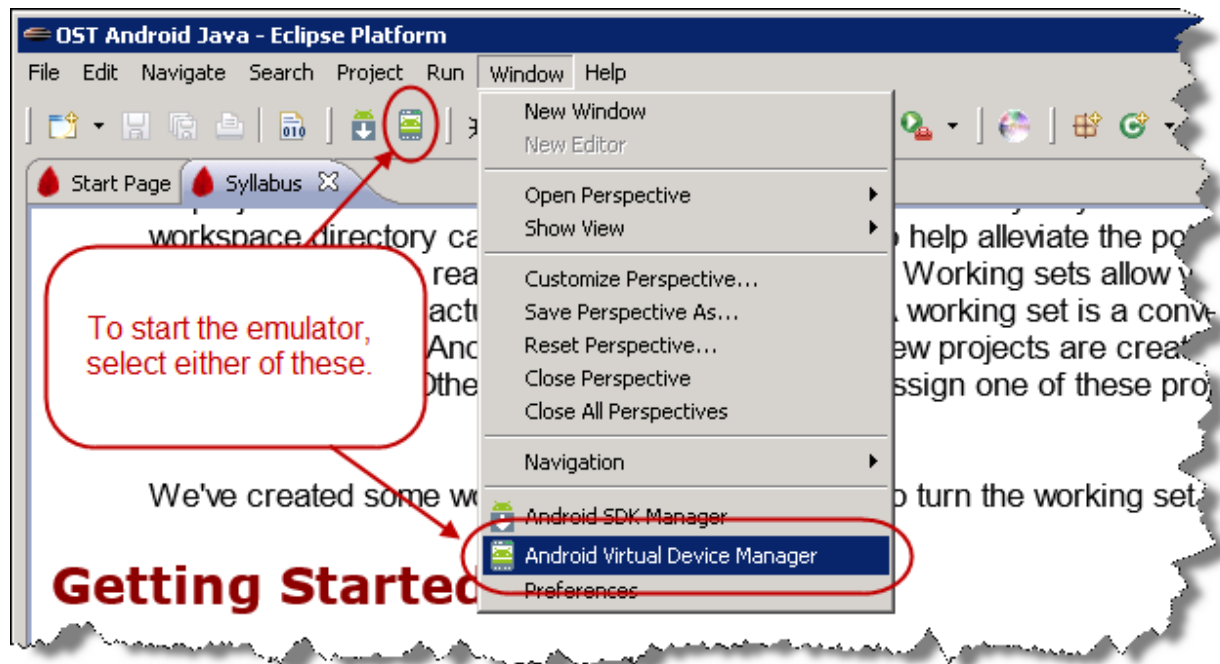
The Android team has made an excellent Eclipse plugin for Android called ADT (Android Developer Toolkit). ADT helps with Android development in Eclipse in many different ways, so it's important that we get the Eclipse environment and ADT set up correctly from the start, so we can build and test our Android applications.

Note

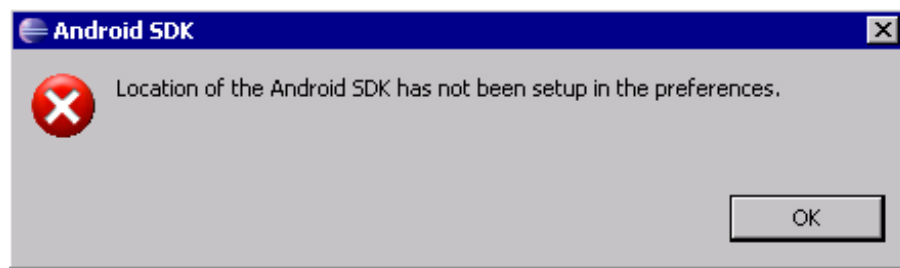
The Android Developer Toolkit plugin for Eclipse changes extremely frequently. The developers behind the toolkit are doing amazing work and constantly updating and improving the plugin. However, this means the most recent version may differ from what you see here and what the instructions detail. Don't worry if what you see slightly differs from the instructions. While the look, feel, and features may have changed (likely for the better), the core decisions and options such as application and package names will generally still be recognizable. We periodically update the toolkit on our systems.

Point ADT to the Android SDK

The ADT plugin is installed on the instance of Eclipse that you are using right now. To open ADT, you can either click the Android Virtual Device Manager icon in the button bar at the top, or select **Window | AVD Manager**:



Go ahead and try that now. You'll probably get an error message informing you that the Android SDK could not be found:

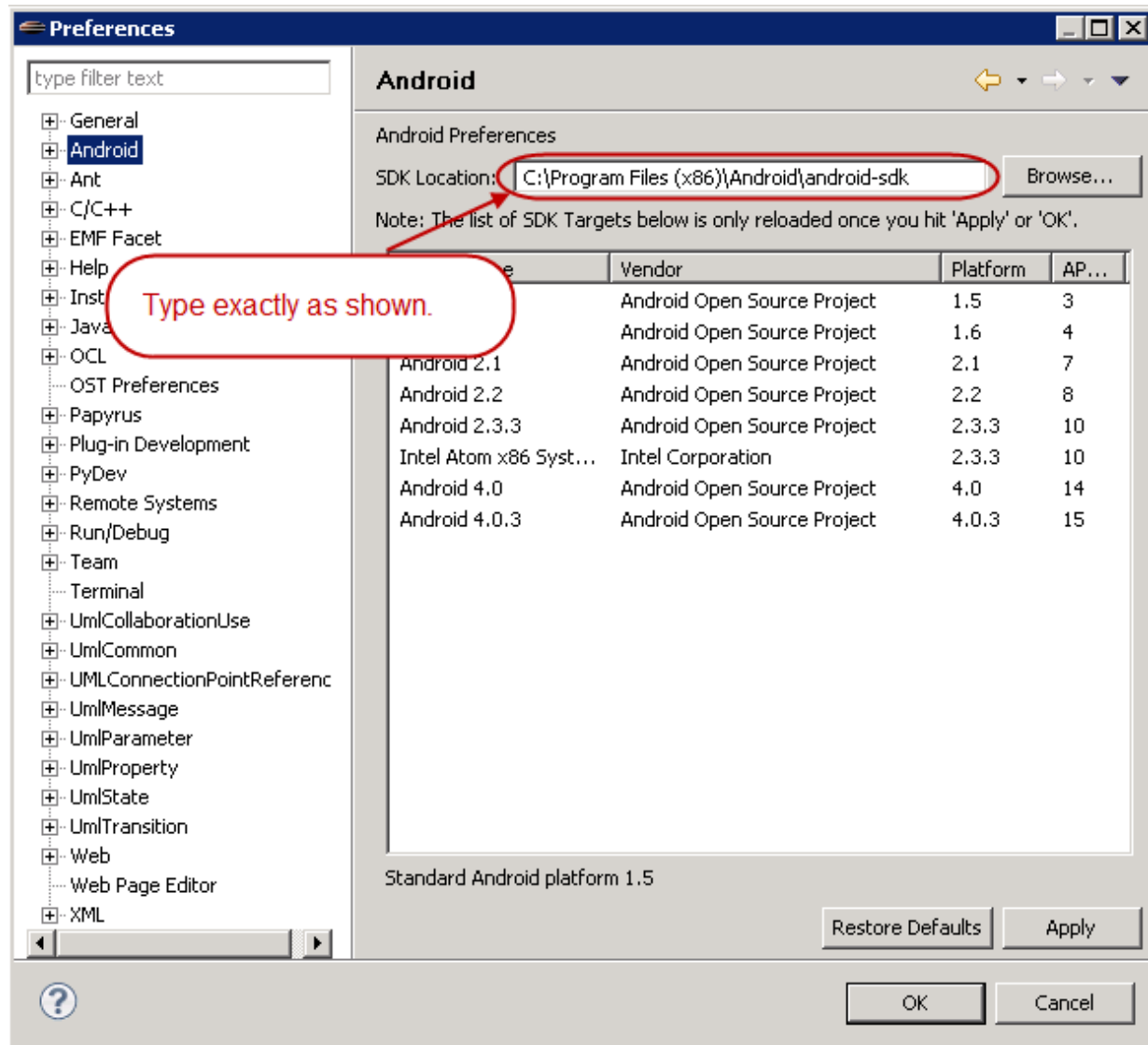



To fix this error, open the Eclipse preferences from the toolbar menu by clicking **Window | Preferences**. The Eclipse preferences window will appear. Then click the **Android** section on the left. (You may be asked if you want to send usage data to Google. Click "No.") Then, in the SDK Location field, type **C:\Program Files (x86)\Android\android-sdk** and click **OK**.

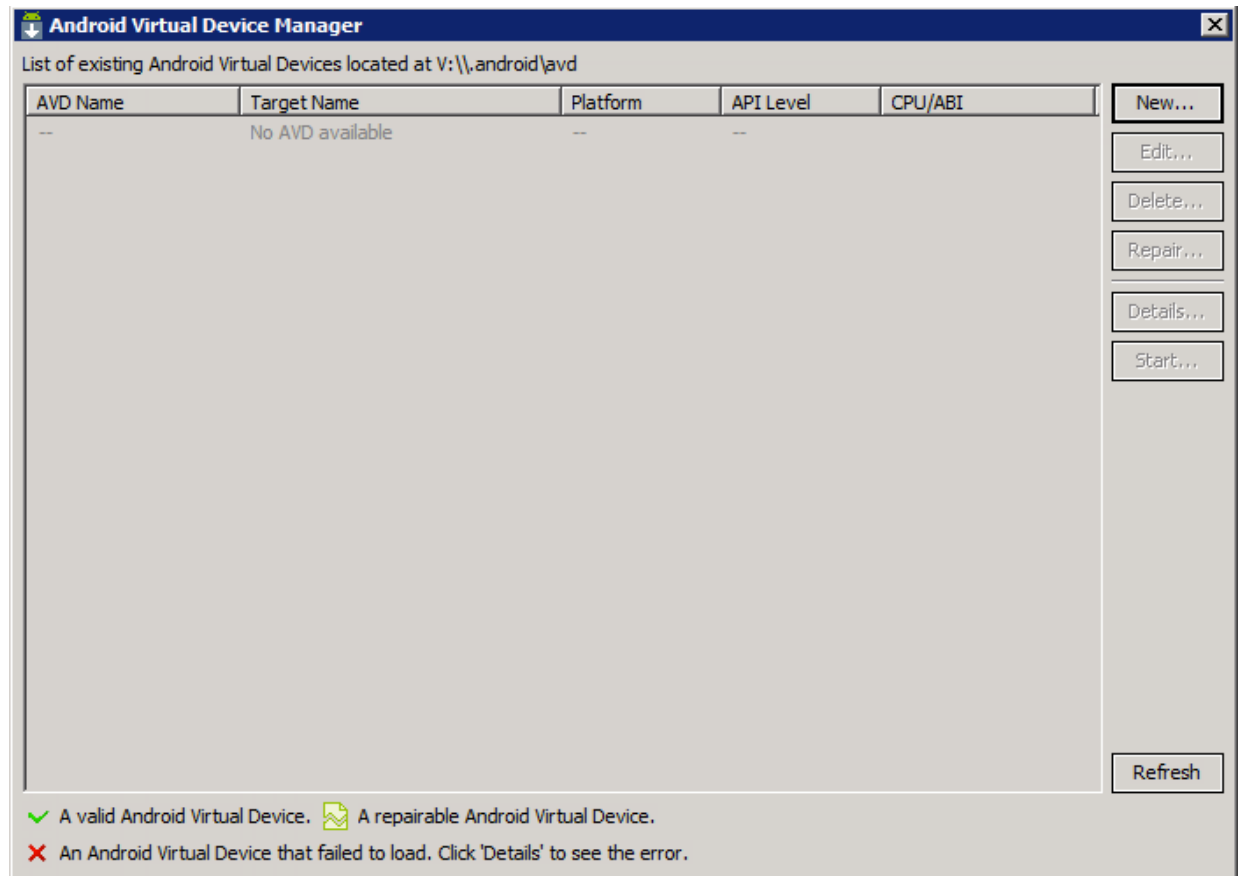
Note

Sometimes when reopening a remote Eclipse session, ADT will forget that it already has the location of the SDK, and will pop-up the error again. If that happens, just open the Eclipse Preferences window again (**Window | Preferences**) and it should show that the path is in there already. Click **OK** and everything should work fine again.

Your Preferences for Android will look like this:



Now ADT is ready to go! To test to make sure it's working, open the ADT window by clicking the  button or selecting **Window | AVD Manager**. The ADT dialog window will open. Feel free to look around in the window to get an idea of what goes on there before you continue on to the next section, where we'll create an emulator using the AVD Manager.



Note

Your AVD Manager probably won't be empty like the screenshot above. Due to the nature of the remote development environment we're using and the way the AVD Manager handles emulators, you'll probably see many other users' emulators. Conversely, any changes you make in the AVD Manager will be visible to other users as well. *Please* be respectful of the other users and *do not* modify or delete any emulators other than those you've created for yourself.

Create an Emulator

If you closed it, open your ADT window again. This is the window that allows you to create and configure as many Android emulators as you like so you can test your application on various different hardware and software configurations. For now, we'll create a single emulator.

On the right side of the ADT window, click **New...** The "Create new Android Virtual Device (AVD)" wizard appears.

- For the Name, enter ***your-ost-username-android2.2.3*** (for example, if your username is **jjamison**, your emulator name would be **jjamison-android2.2.3**).
- In the Device dropdown, select the **Nexus S**.
- in the Target dropdown, select **Android 2.2.3 - API Level 10**.
- For the SD card, select the **Size** radio button and enter **20** MiB.

Create new Android Virtual Device (AVD)

AVD Name:

Device:

Target:

CPU/ABI:

Keyboard: ☒ Hardware keyboard present

Skin: ☒ Display a skin with hardware controls

Front Camera:

Back Camera:

Memory Options: RAM: VM Heap:

Internal Storage:

SD Card:

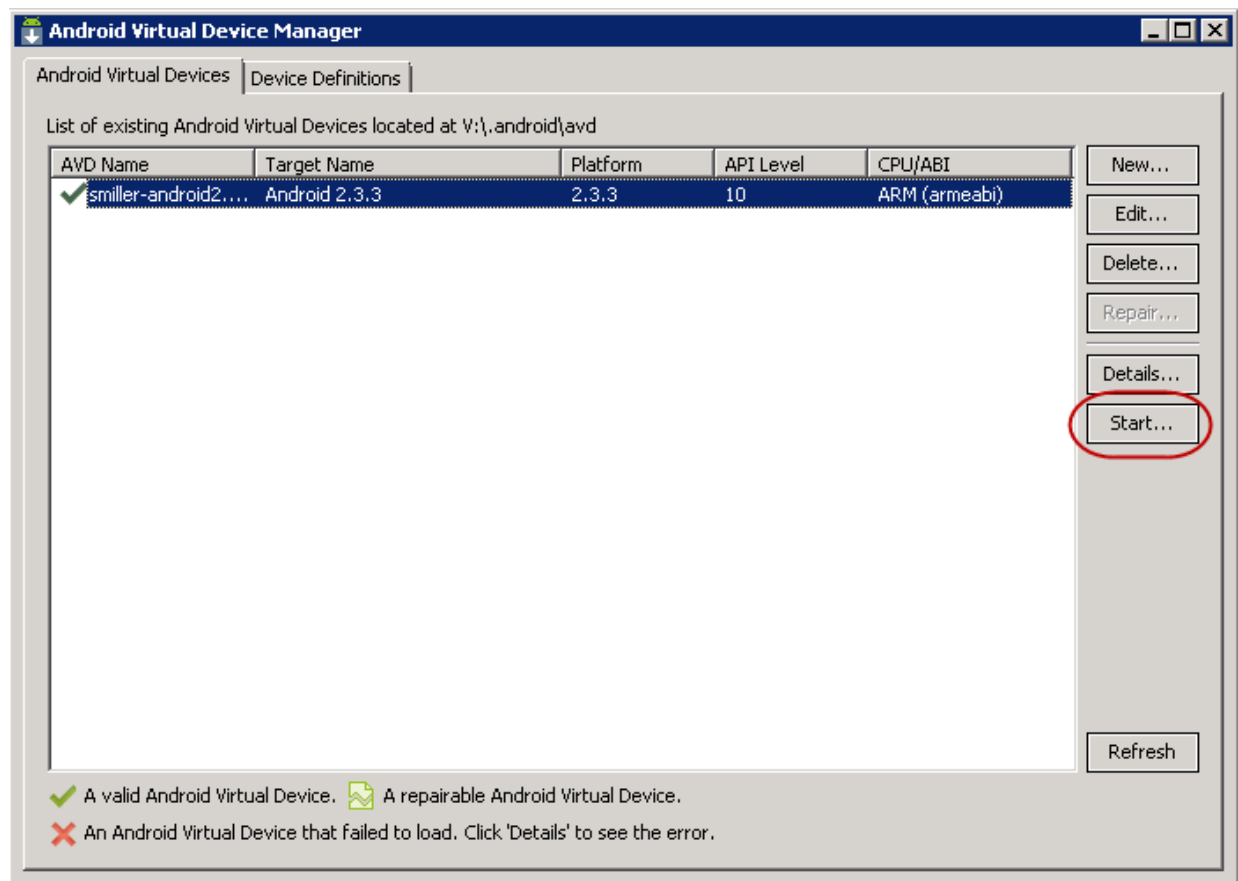
☒ Size:

☐ File:

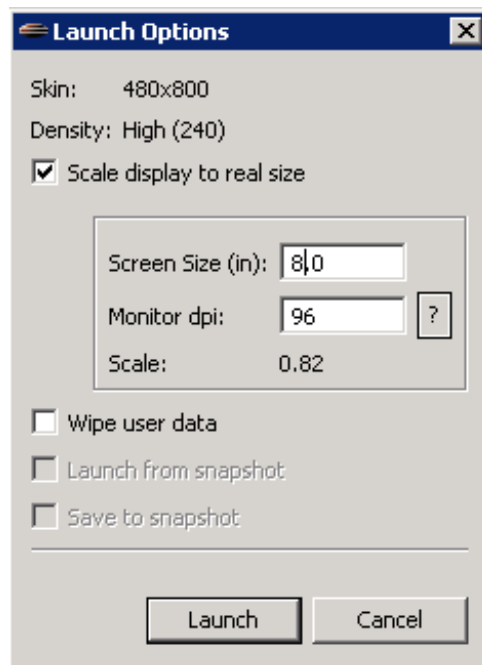
Emulation Options: ☐ Snapshot ☐ Use Host GPU

☐ Override the existing AVD with the same name

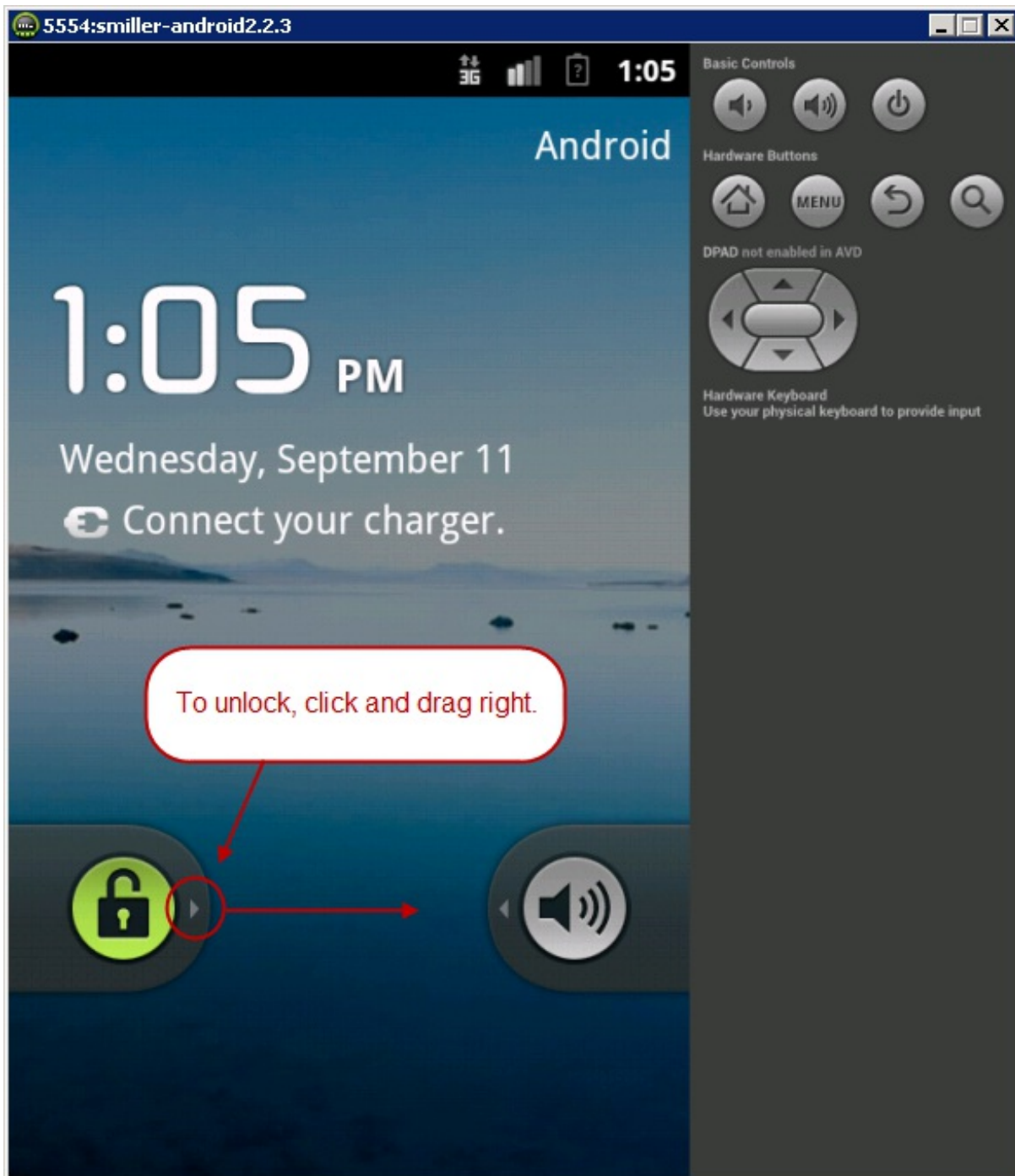
When you're ready, click **Create AVD** at the bottom. Then, select your new emulator in the Virtual Devices list, and click **Start...** on the right:



A Launch Options window appears. The emulator is actually a little too big for our remote Eclipse session, so we'll scale it down a little. Check the **Scale display to real size** box, enter **8.0** in the Screen Size (in.) field, and then click **Launch**:

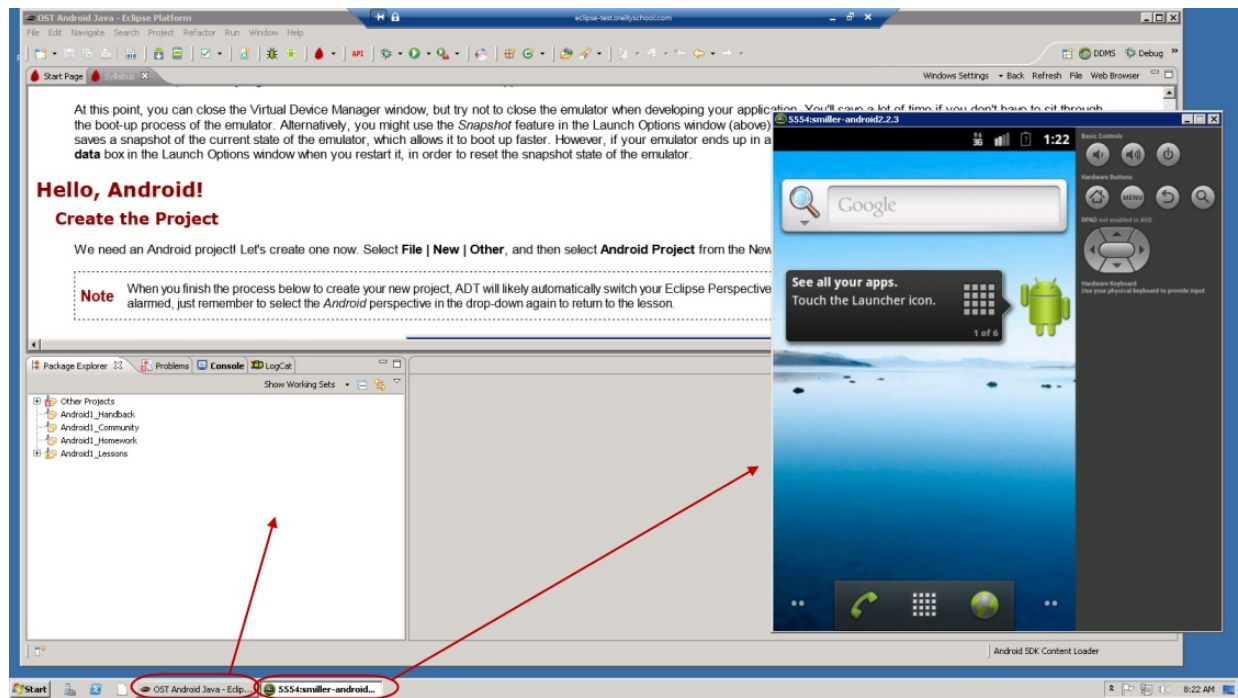


The emulator will take a while to load. Now might be a good time to pour yourself another cup of coffee or let the dog out. When the emulator is finally loaded, you'll see it in another window on top of Eclipse.



At this point, you can close the Virtual Device Manager window, but try not to close the emulator when developing your application. You'll save a lot of time if you don't have to sit through the boot-up process of the emulator. Alternatively, you might use the *Snapshot* feature in the Launch Options window (above). In Snapshot mode, whenever the emulator is closed, AVD saves a snapshot of the current state of the emulator, which allows it to boot up faster. However, if your emulator ends up in a weird or broken state, you'll need to check the **Wipe user data** box in the Launch Options window when you restart it, in order to reset the snapshot state of the emulator.

To switch between this lesson content and the emulator, use the tabs at the bottom of the screen:



Note

You can set up other emulators to match different devices, if you like. Always begin the emulator name with your OST user name, so you can differentiate them from emulators created by other users.

In the next section, we'll finally dig into some code and run our first Android application!

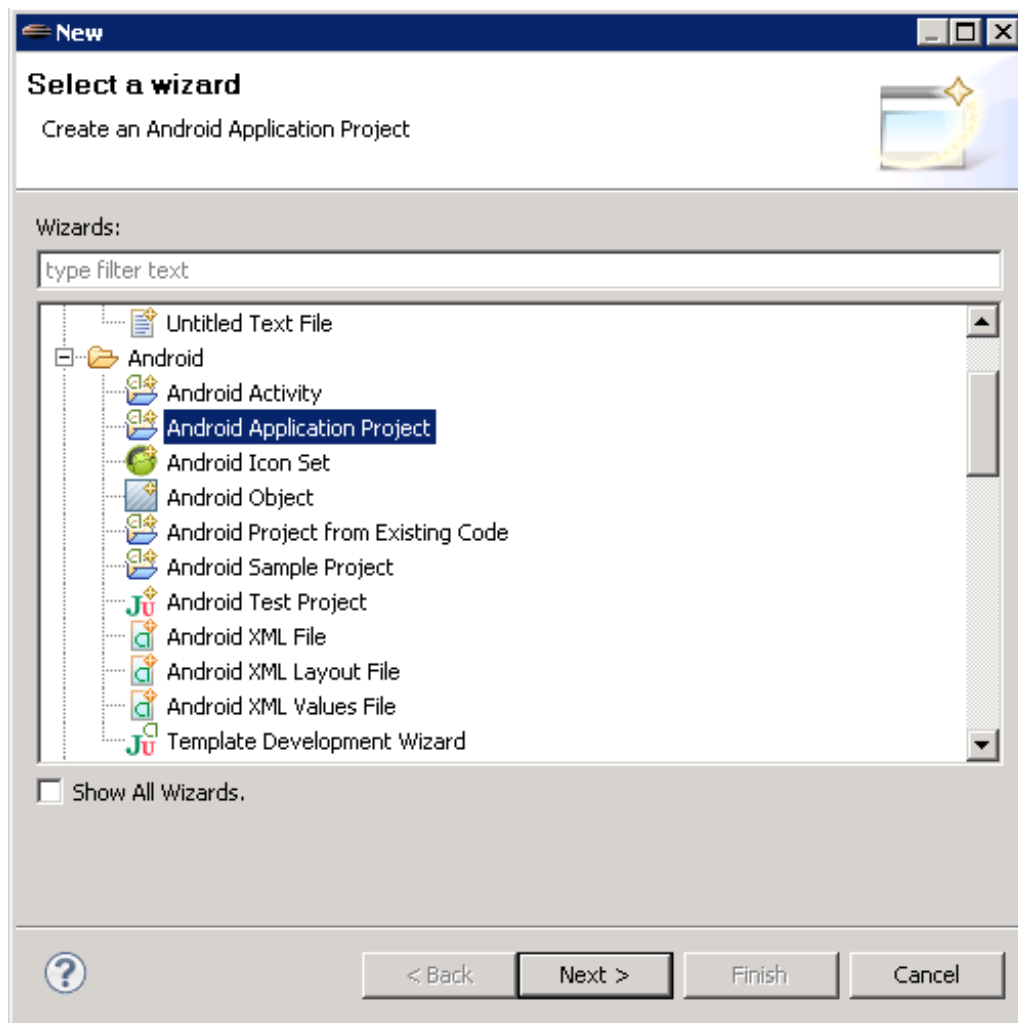
Hello, Android!

Create the Project

We need an Android project! Let's create one now. Select **File | New | Other**, and then select **Android Application Project** from the New Project Window as shown below:

Note

When you finish the process below to create your new project, ADT will likely automatically switch your Eclipse Perspective to Java (which will hide this instruction window). Don't be alarmed, just remember to select the *Android* perspective in the drop-down again to return to the lesson.



Now you see the first window of the "New Android Application" Wizard. This process takes you through three different windows to help set up your new project. In the first window, type the Project name as **HelloWorld**, enter the Package Name **com.ost.android1.hello world**, and select the other options as shown:

New Android Application

Creates a new Android Application

Application Name: HelloWorld

Project Name: HelloWorld

Package Name: com.ost.android1.helloworld

Minimum Required SDK: API 10: Android 2.3.3 (Gingerbread)

Target SDK: API 10: Android 2.3.3 (Gingerbread)

Compile With: API 10: Android 2.3.3 (Gingerbread)

Theme: None

Choose the base theme to use for the application

< Back Next > Finish Cancel

Click **Next**. In the next window, uncheck the **Create custom launcher icon** box, and make sure the **Add project to working sets** box is checked and the **Android1_Lessons** working set is entered in the Working Sets field:

New Android Application

New Android Application
Configure Project

☐ Create custom launcher icon

☒ Create activity

☐ Mark this project as a library


☒ Create Project in Workspace

Location:

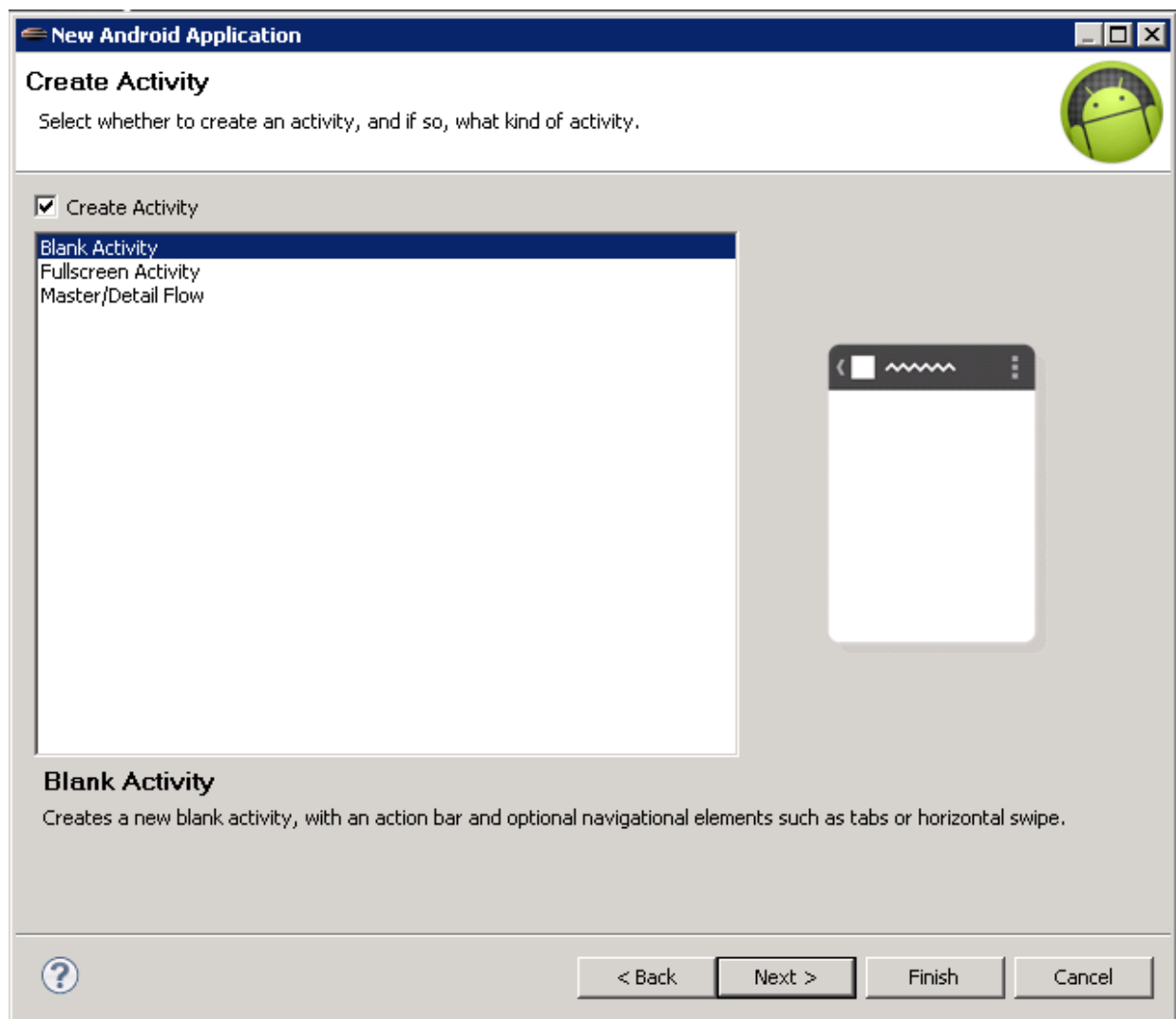
Working sets

☒ Add project to working sets

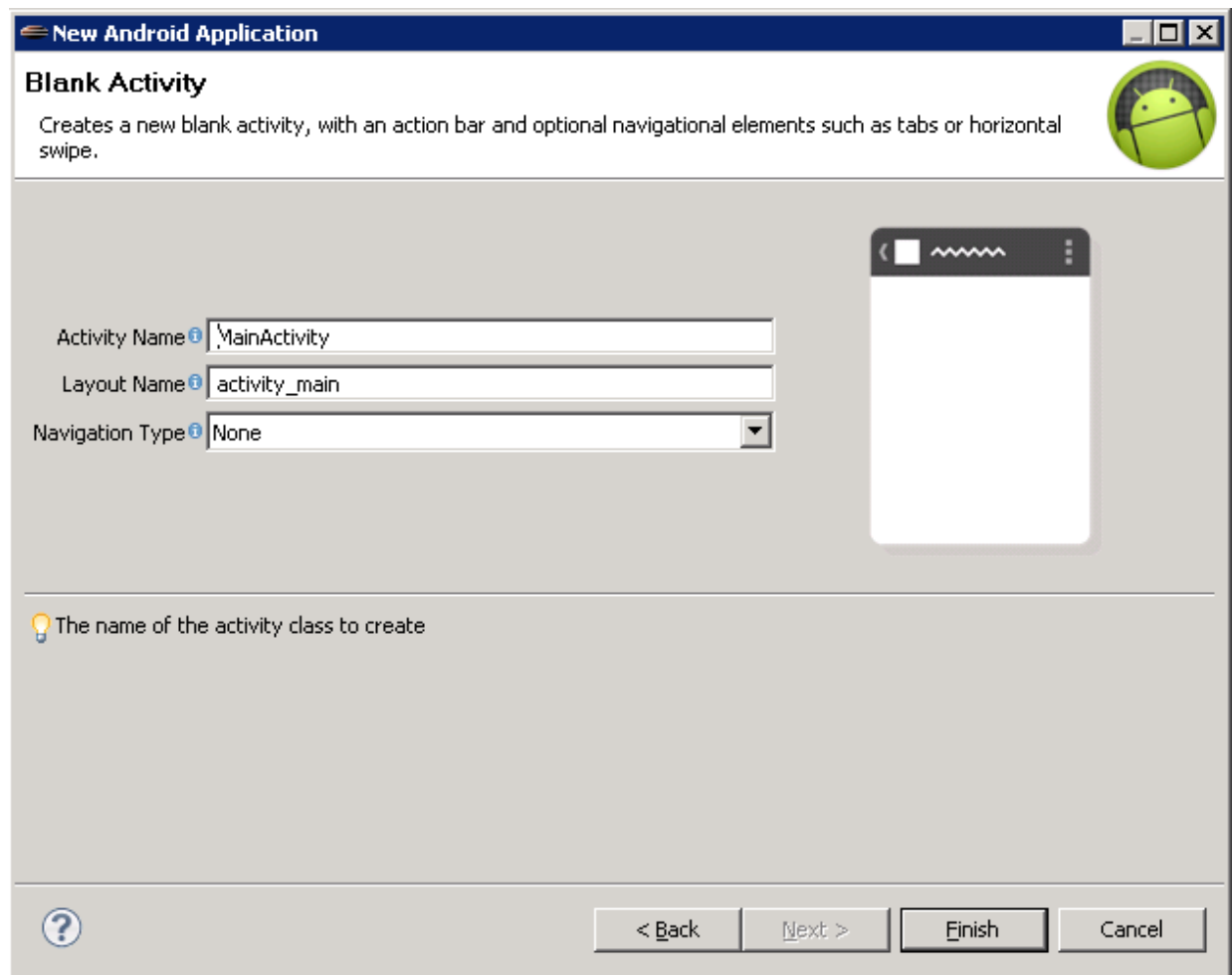
Working sets:



Click **Next**. In the next window, check the **Create Activity** box and select **Blank Activity**:



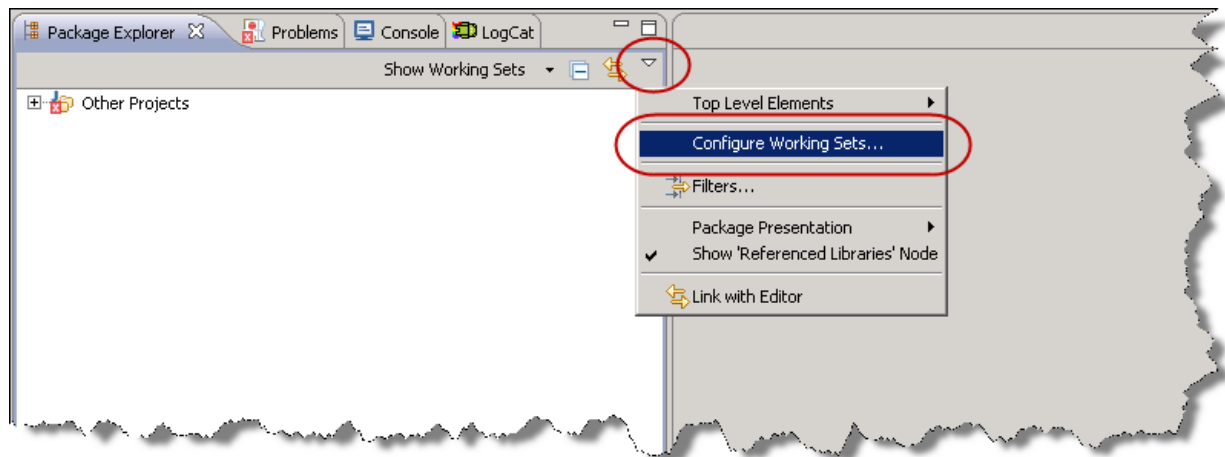
Click **Next**. In the next window, accept the default Activity Name **MainActivity**, Layout Name **activity_main**, and Navigation Type **None**:



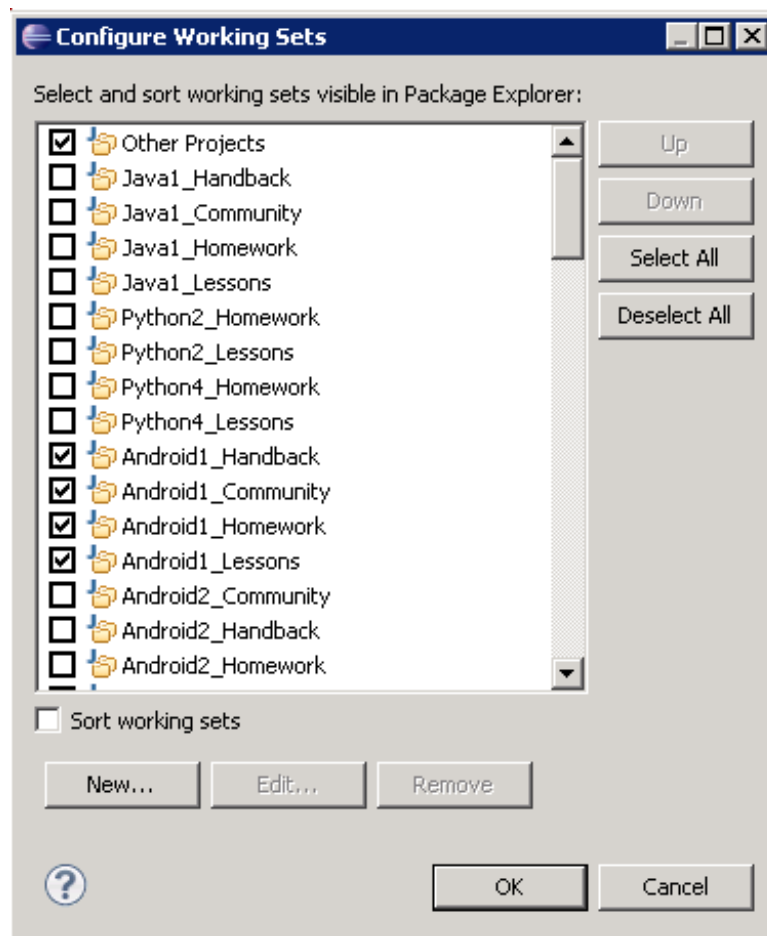
Click **Finish**.

Remember these steps—you'll need to perform them for any new project you create in this course.

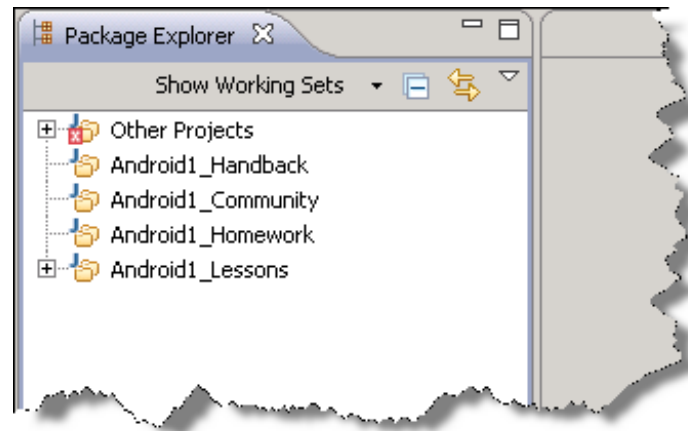
If **Android1_Lessons** doesn't appear in your Package Explorer window, fix it now. In the top-right corner of the Package Explorer window, click the downward-pointing arrow and select **Configure Working Sets....**



Check the boxes for the **Android1** working sets and click **OK**:

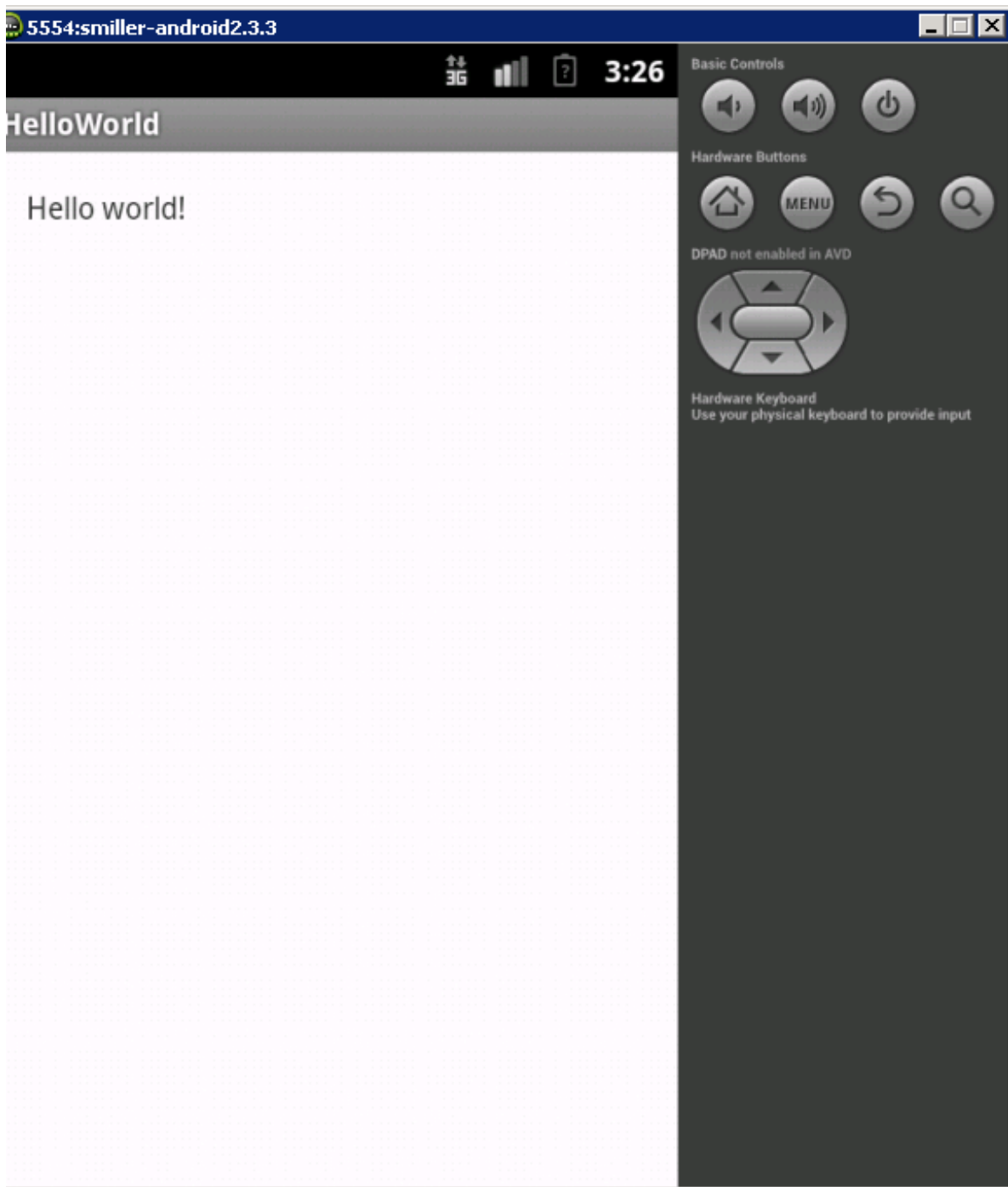


Now you'll see those working sets (and the Other Projects) in the Package Explorer window:



Run the Application

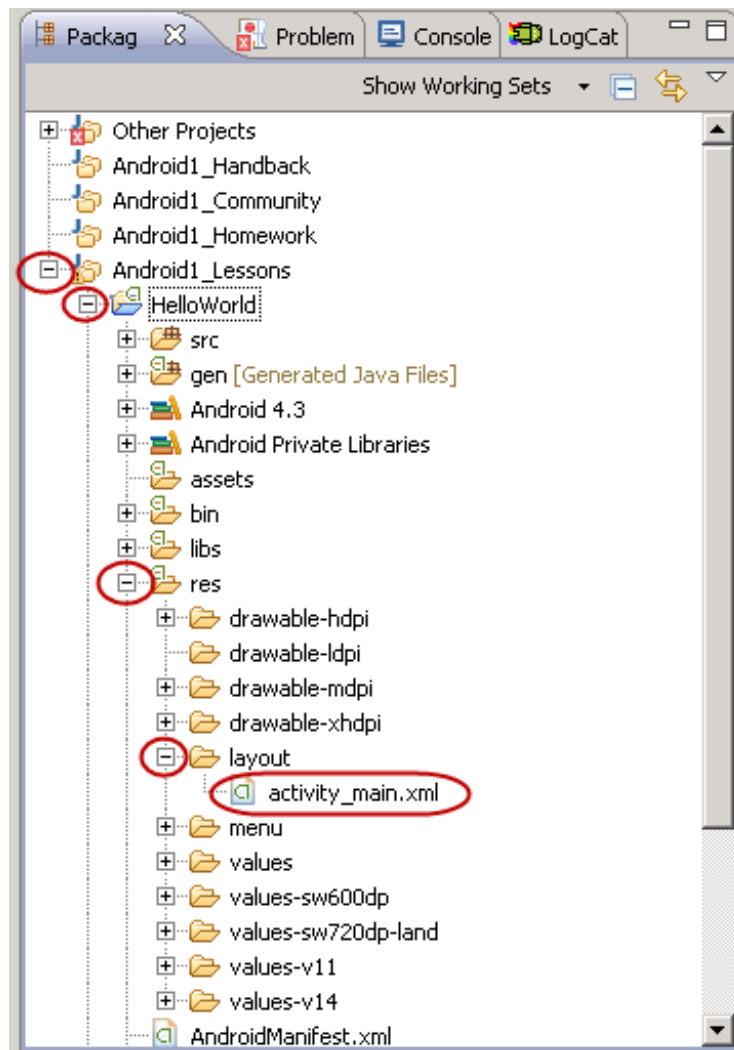
To run the application, right-click the root project folder **HelloWorld** in the Package Explorer, and select **Run As | Android Application**. If your emulator was closed, it will open automatically now; if it was still open, you'll have to bring the emulator window back to the front. If your emulator is in lock mode, unlock it by dragging the green unlock button to the right side of the screen. Once ADT has finished installing the application onto the emulator, it will launch automatically.



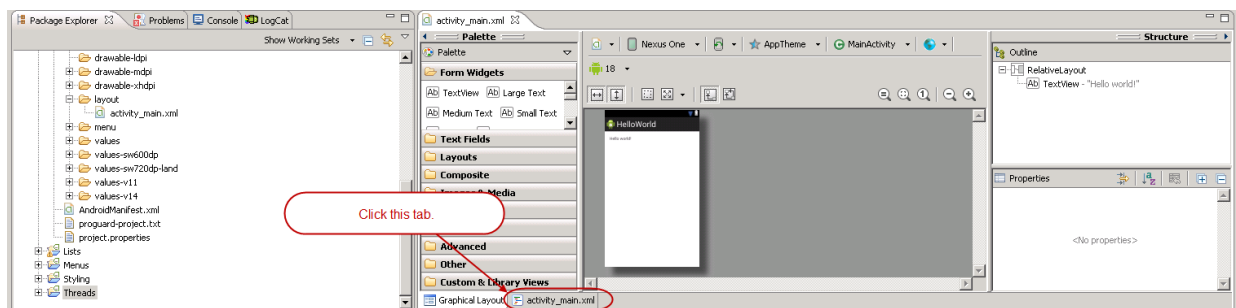
It's not much to look at yet, but it's a great start. Now we have a solid foundation to start getting into some real Android application development.

Editing Programs

When you create the project, the **activity_main.xml** file, in the **/res/layout** folder, is created:



By default, Android XML files load in a *Graphical Layout* view. We'll talk about that in detail later; for now, we'll focus on the actual XML. Click the **activity_main.xml** tab in the lower portion of the editor screen:



Edit the code as shown:

CODE TO TYPE:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_worldHello World!" />

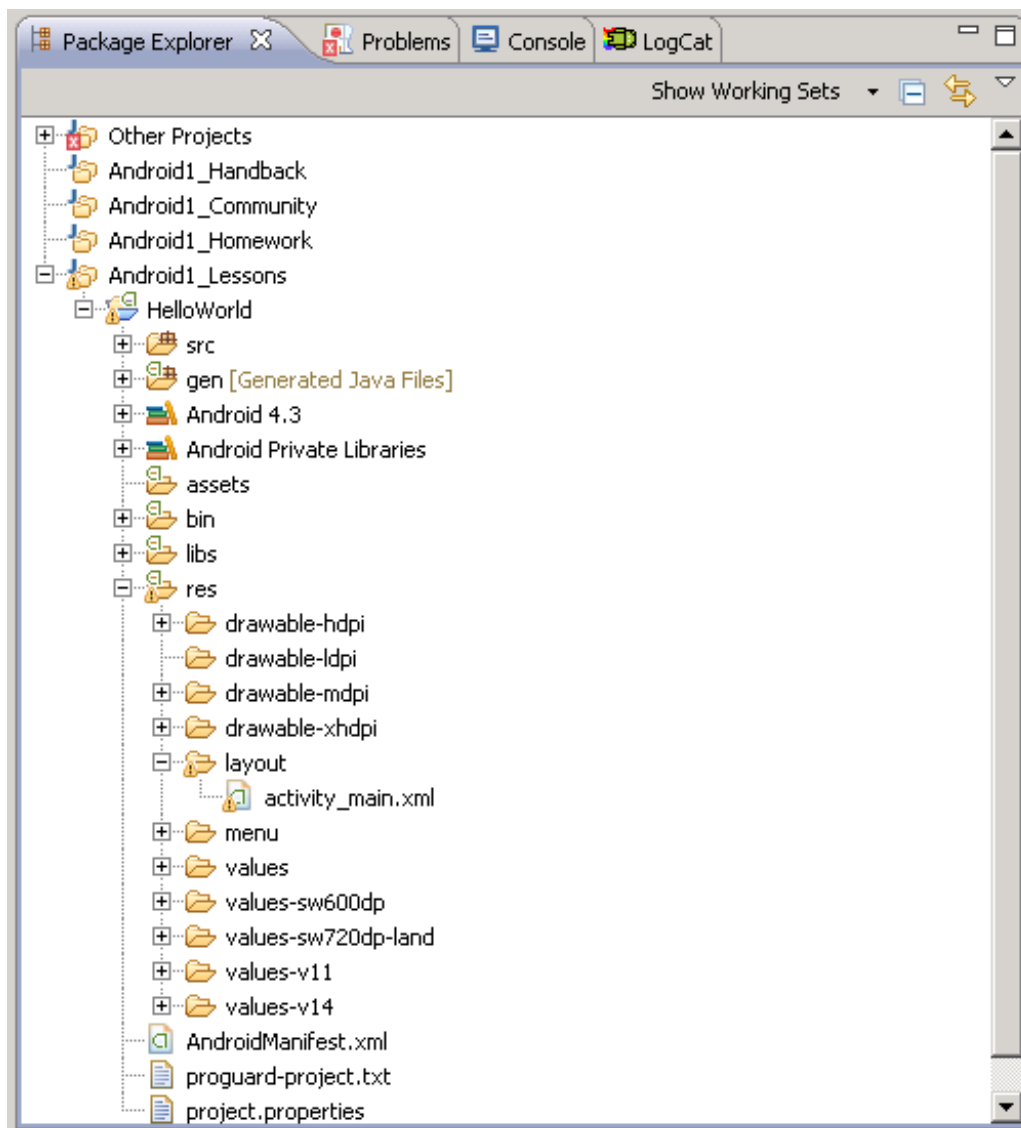
</RelativeLayout>
```

Save and run the application again. You see your new text:

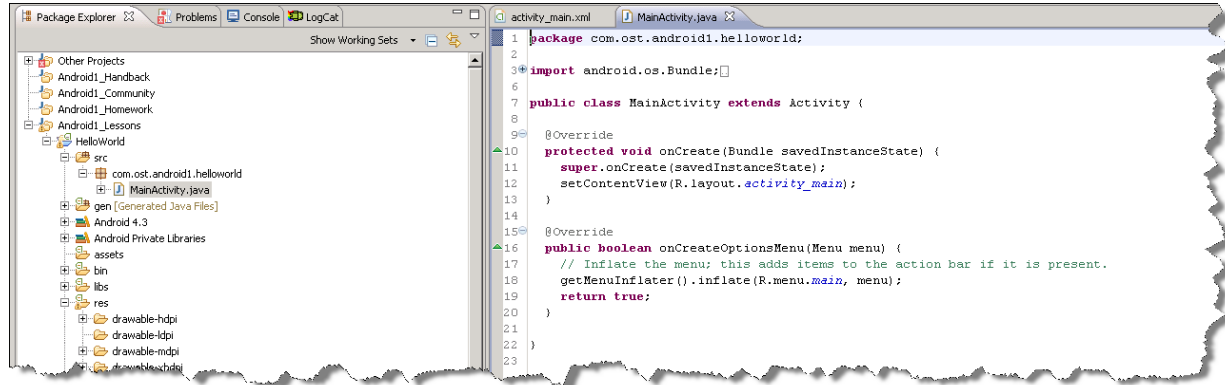
□

Android Package Structure

Let's take a moment to get familiar with the Android package structure. Take note of the default files that were created in the root directory of the project:



All Android projects have an **AndroidManifest.xml** file, along with a **/res** folder, and a source folder, usually titled **/src**. If you open the **/src** folder, and then the **com.ost.android1.helloworld** package, you'll see the **MainActivity** class that we defined when creating the project. Go ahead and double-click that file to open it now:



Let's take a look at the code:

OBSERVE: MainActivity.java

```
package com.ost.android1.helloworld;

import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;

public class MainActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    public boolean onOptionsItemSelected(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }
}
```

This MainActivity is the first entry point into our Java code for this application. The **onCreate()** method is first called when the Activity is created. We will cover the Activity class in depth in the next lesson, but for now, just be aware that each view in an application is controlled by an Activity.

Also, notice that the second line of **onCreate()** calls **setContentView(R.layout.activity_main)**. This method loads the view that MainActivity will control. **R.layout.activity_main** is a reference to the **activity_main.xml** file in the **/res/layout** folder. Let's look at that file again:

OBSERVE: activity_main.xml

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity" >

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Hello World!" />

</RelativeLayout>
```

It may seem like there's a lot going on in this method, but we'll just focus on the tag names for now. This view defines a **RelativeLayout** with one child, a **TextView**.

Bonus Round

Haven't had enough yet? That's great! There is so much more we can do now that we have a running application. Let's get back into the code and start making some changes of our own!

Our earlier change was pretty straightforward. Let's try changing it up a bit more. Edit **activity_main.xml** again as shown:

/res/layout/activity_main.xml

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context=".MainActivity" >

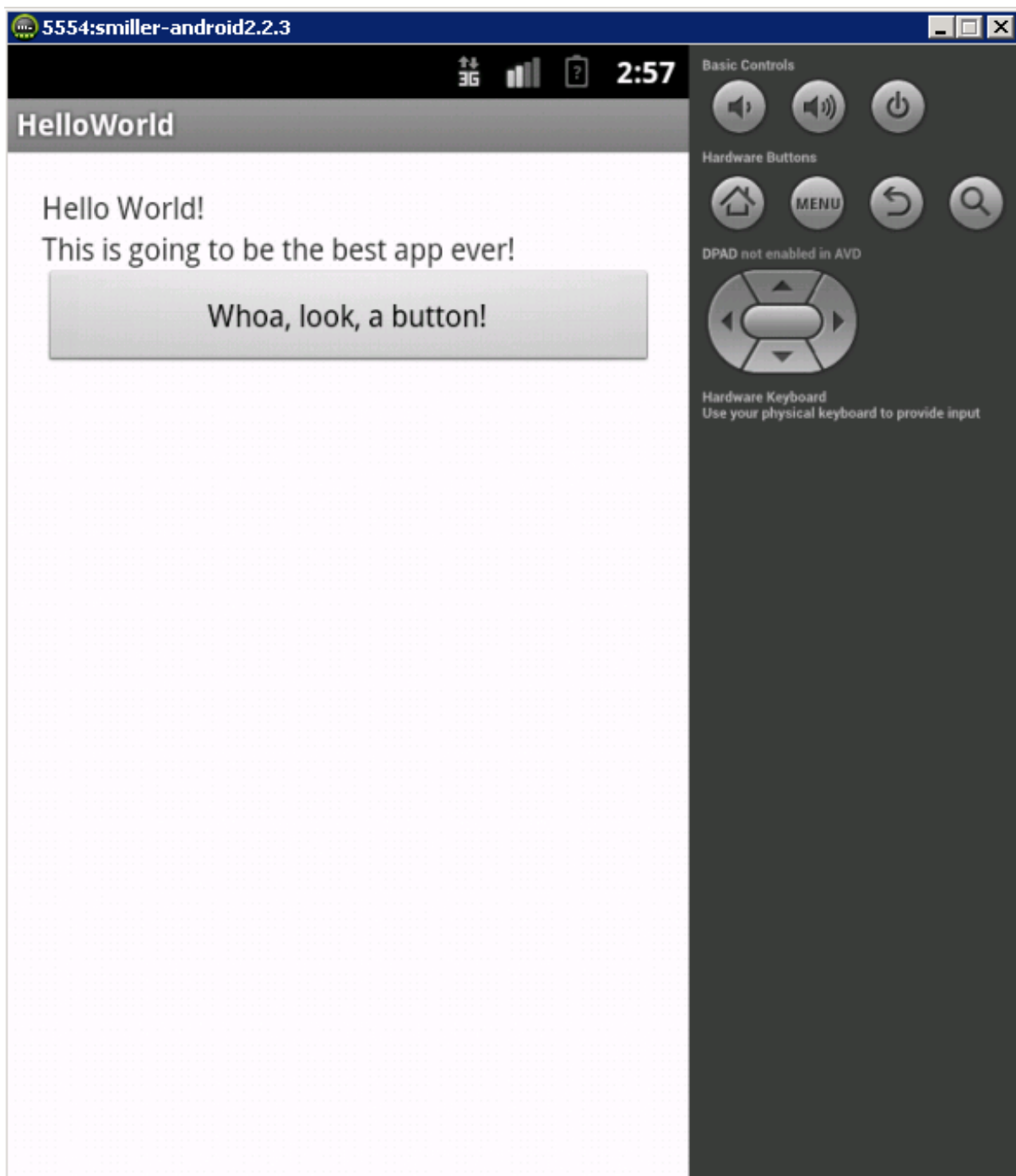
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Hello World!" />

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="This is going to be the best app ever!" />


    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Whoa, look, a button!" />

</RelativeLayout>
```

Save and run it again to see your view has grown:



Note

After you run an application for the first time using right-click and **Run As** or the **Run** menu, there's a faster way to run it. You can click the Run icon button  in the button bar at the top. With Eclipse, there's often more than one way to accomplish a particular task. These shortcuts will help cut down on your development time, so you'll definitely want to use them!

Wrapping Up

We've covered lots of topics here that are essential to every good Android developer. From setting up your environment to creating an emulator to creating and running an Android project, these skills form the foundation for building and testing any Android application. You're doing great—see you in the next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Activities and Views

Welcome back! In the last lesson we covered the foundations of Android development—setting up the Eclipse environment with ADT, creating an emulator, creating a new Android project, and installing and running it on the emulator. In this lesson we'll learn more about views, and also explore the fundamental classes of every Android application.

AndroidManifest.xml

Every Android project must have an AndroidManifest.xml file located in the root of the project directory. Think of it as the backbone of your Android application, defining the package name (unique for each application in the market), every Activity and Service, each permission that the Application requires, and more. We'll refer back to the AndroidManifest often during the course.

Let's go back into our existing project and use it to demonstrate the importance of the AndroidManifest. We'll start by writing some code to launch a new activity. Open your project, then open the **MainActivity.java** file and then edit the code as shown:

MainActivity.java

```
package com.ost.android1.helloworld;

import android.app.Activity;
import android.os.Bundle;
import android.content.Intent;
import android.view.Menu;

public class MainActivity extends Activity {
    @Override

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        startHermes();
    }

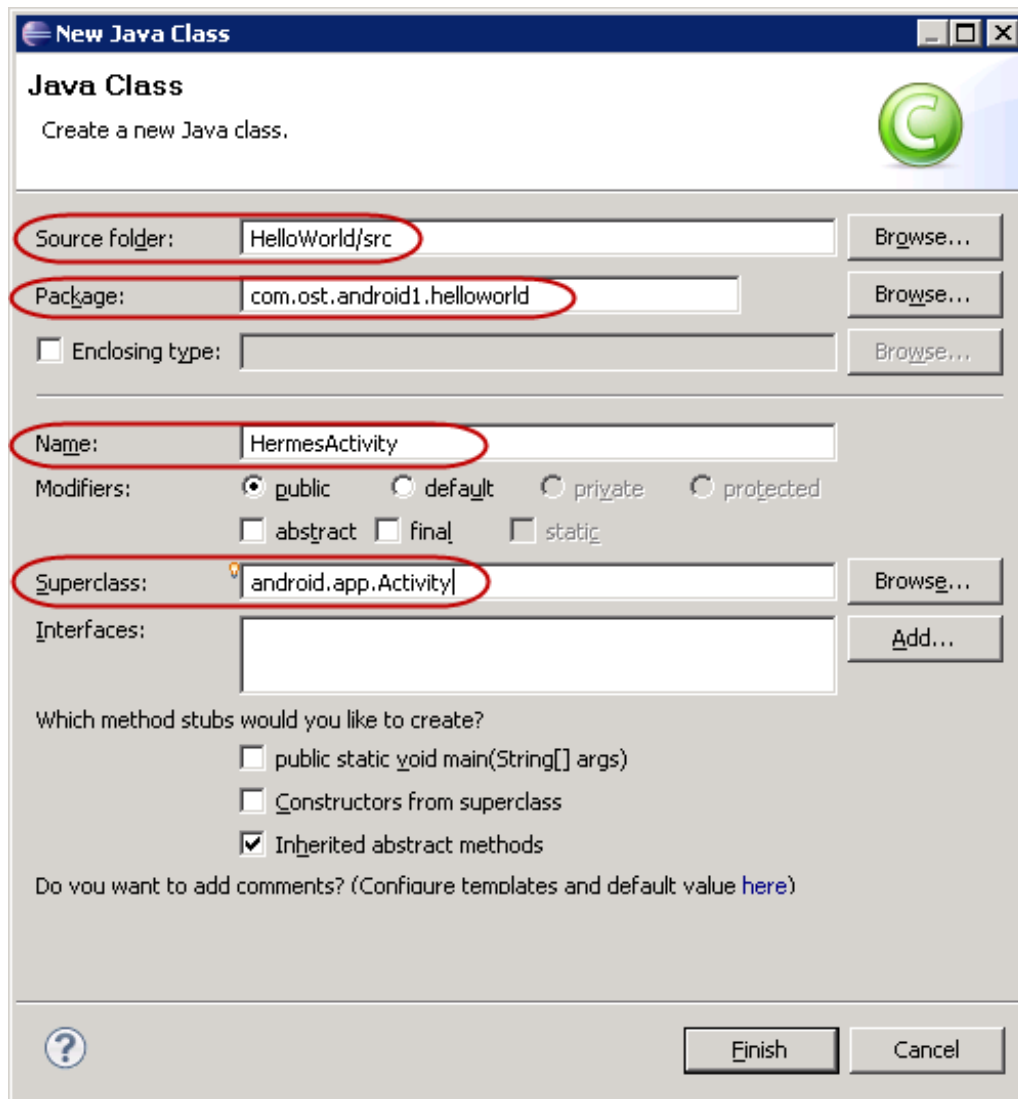
    public void startHermes() {
        Intent intent = new Intent(MainActivity.this, HermesActivity.class);
        startActivity(intent);
    }


    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }
}
```

Note

The Importance of import: If you're familiar with Java, then you know the importance of import statements at the top of classes. Throughout the course we'll reference other classes that will require additional imports (such as the Intent class above, which requires the import declaration *import android.content.Intent*;). I will rarely refer to the exact import updates that are necessary for each code change we make, though, because Eclipse can add those imports to our files automatically. There are various ways to get Eclipse to do this. For example, you might use **Source | Organize Imports** on the menu or the keyboard shortcut **Ctrl+Shift+O**. These commands will save you a lot of development time in Java.

The code we've just written will launch a new Activity called *HermesActivity* during the onCreate method. (The Intent class is an important one in Android and it has many purposes beyond launching Activities; we'll discuss those other purposes in detail a bit later.) Eclipse displays a red squiggly line under *HermesActivity.class*, because it doesn't exist yet. Let's create it now. Select **File | New | Class**. In the "New Class" window, name the class *HermesActivity* and set the superclass to be **android.app.Activity**:

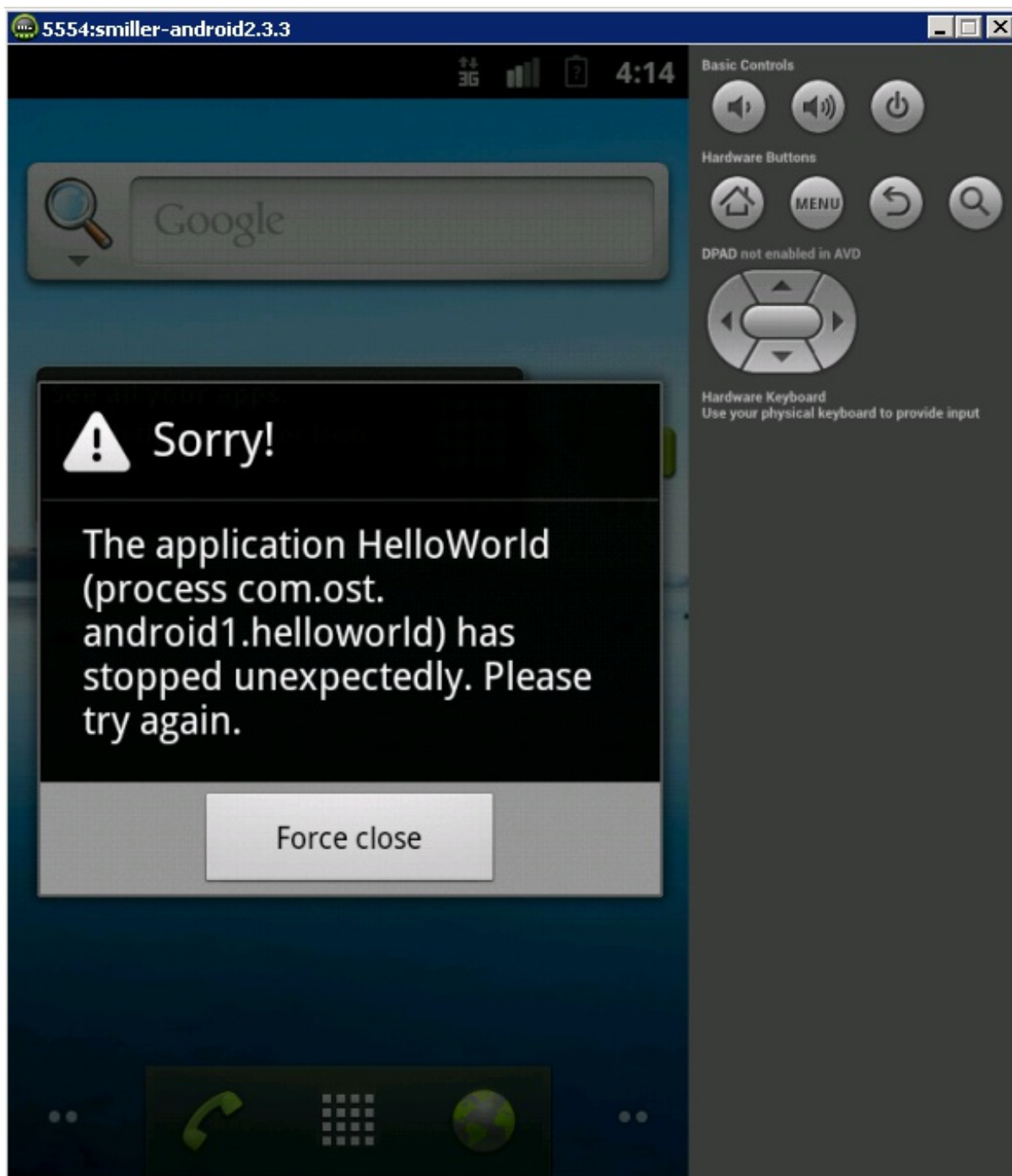


Click **Finish** to create the class. You can close the HermesActivity.java class file now, because we won't be modifying it for this demonstration. Now let's run the application (click the **Run**  icon).

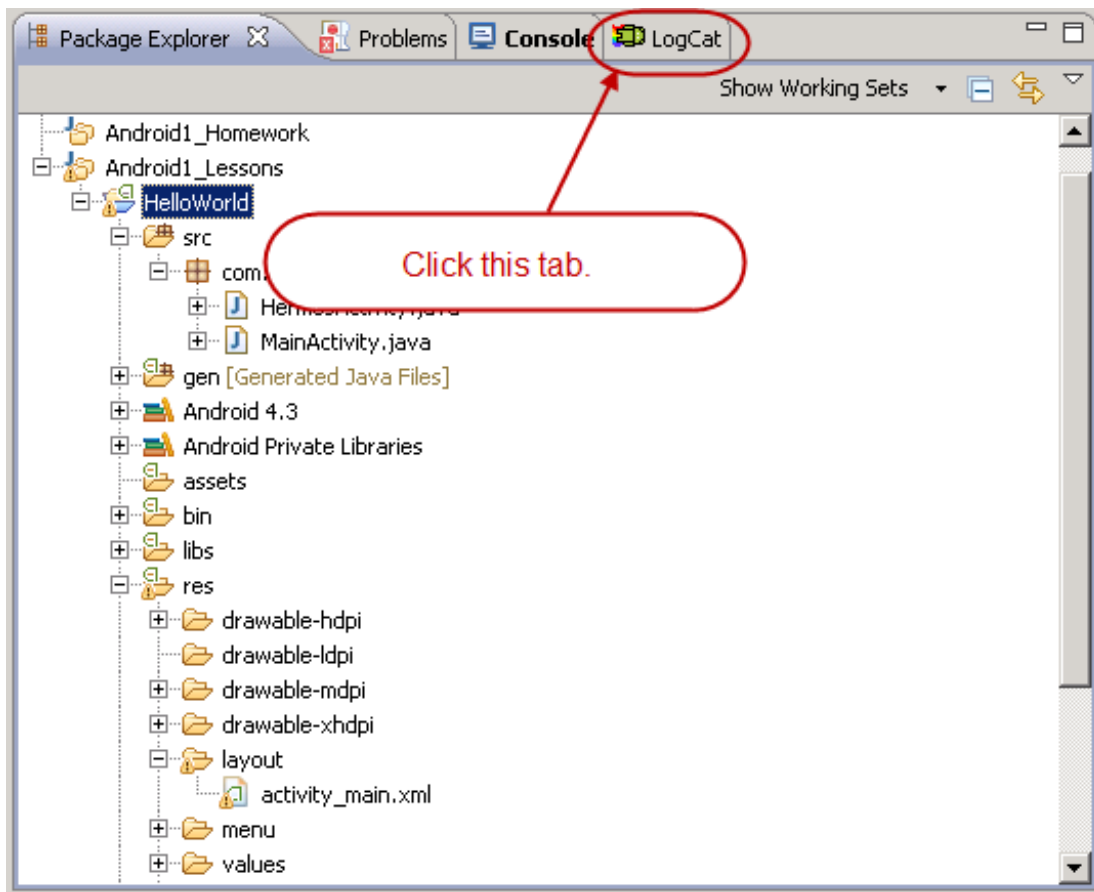
Note

The emulator should start up automatically if it wasn't already started before running the project. It may take a while for it to start though. When it finishes, the project should install on to the emulator automatically and execute. Although on occasion Android may think the emulator has timed-out while waiting for it to start; if that happens, just re-run the project after the emulator is up and running.

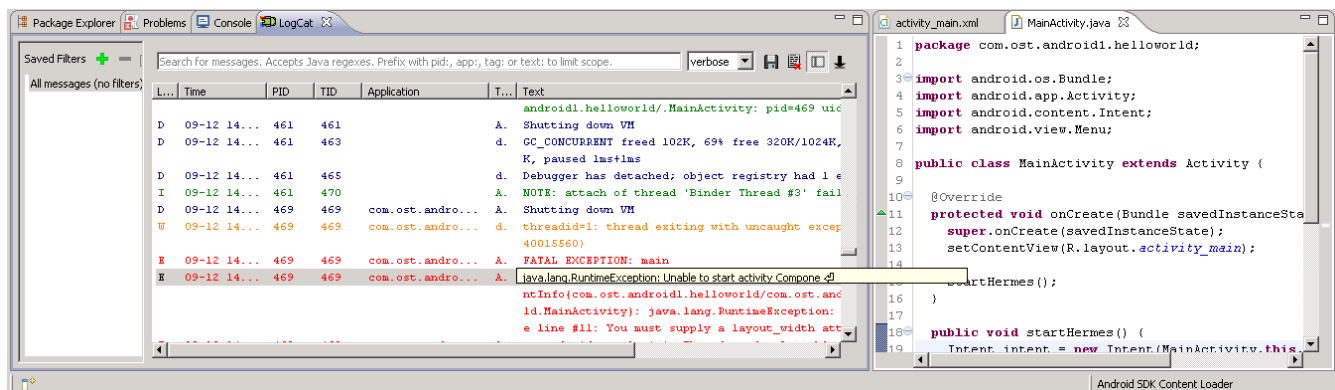
So, how does it look? Did you get an error message that looked something like this?




Don't worry. This was one of those planned errors we sneak in from time to time to get you used to encountering them—and fixing them. This was the most common error I ran across when I first dove into Android programming. Let's take a look at the logs and see what's going on. Click the **LogCat** view tab on the Package Explorer pane:



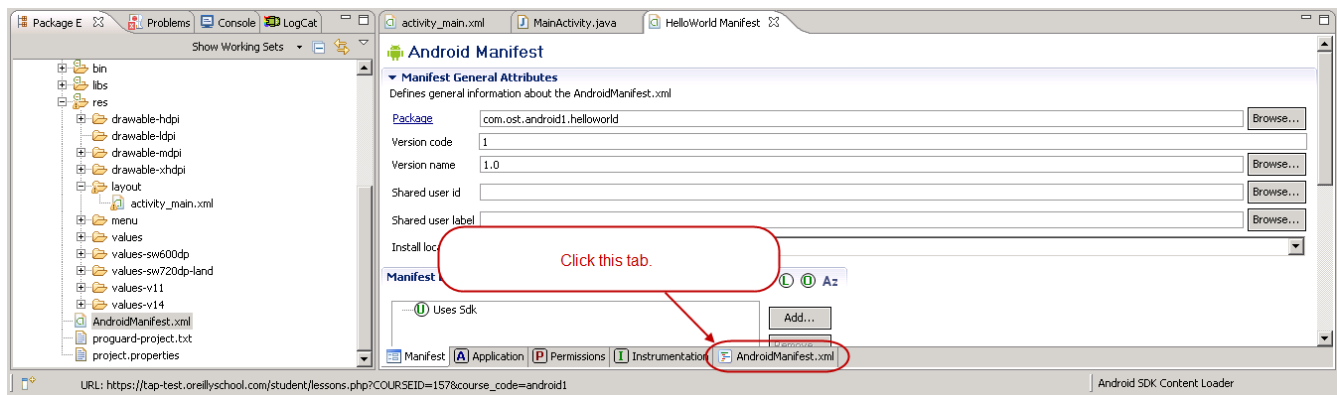
This view displays all the log information from a connected emulator or device. Scroll down to the bottom of the LogCat view and find the red text (the color used for Error logs). Click and drag the right edge of the panel to widen it, and hover with the mouse over the second error. You should see something like this:



Note

You can always have multiple devices connected and multiple emulators running, but *LogCat* can only display one device or emulator's logs at a time. If LogCat isn't showing you the logs you expected, use the DDMS perspective  to select the correct device/emulator. We'll cover the DDMS perspective in detail in a later lesson.

This error is a little vague, but in short, it's saying it couldn't find our Activity, *HermesActivity*. That's because we haven't defined it in *AndroidManifest.xml* yet. As we discussed earlier, the *AndroidManifest* defines each Activity available to an application. The declaration informs the Android system which Activities are present and how they can be launched. So let's add the declaration now. Go back to the Package Explorer tab (drag its right border back to make it narrower) and open **AndroidManifest.xml** in the **HelloWorld** project. There are a lot of different sub-screens available for *AndroidManifest* to help modify the file using a GUI, but we'll just edit the XML directly. Click the **AndroidManifest.xml** sub-tab in the bottom of the view :



Now that we're in the correct view, let's add the HermesActivity to the manifest:

CODE TO TYPE:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.ost.android1.helloworld"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="10"
        android:targetSdkVersion="10" />

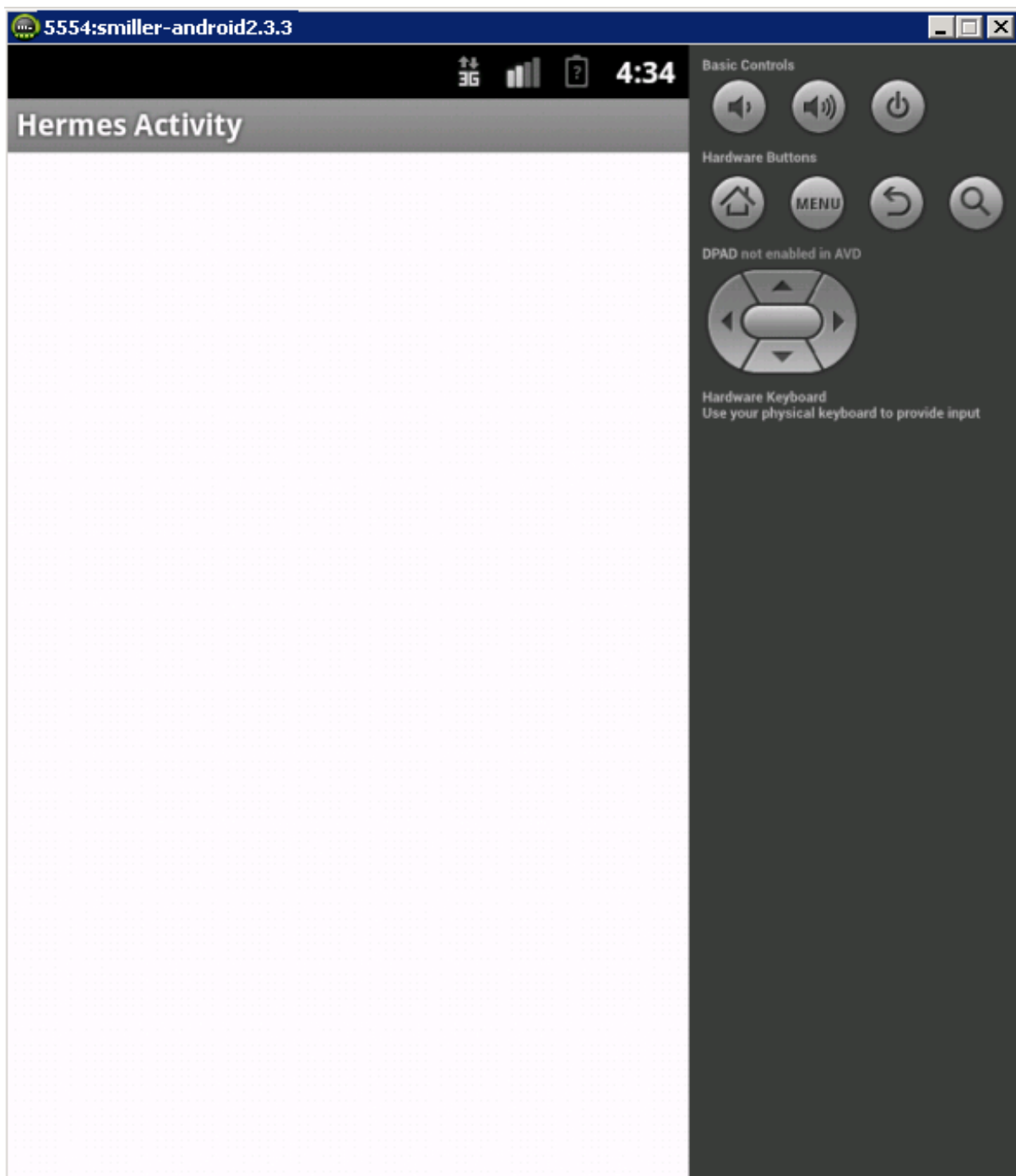
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >

        <activity
            android:name="com.ost.android1.helloworld.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <activity android:name=".HermesActivity"
            android:label="Hermes Activity"/>

    </application>
</manifest>
```

Save the changes and run the application again. This time you don't see the error—or much of anything else:



This probably doesn't come as a surprise, though, because we didn't even add a view to our HermesActivity.

Let's analyze the changes we did make to the AndroidManifest:

OBSERVE:

```
<activity android:name=".HermesActivity"
  android:label="Hermes Activity"/>
```

Each Activity in an Android application requires an `<activity />` node, nested within the `<application />` node. `android:name` refers to the name of the activity prefaced by the package in which the Activity is located, relative to the package of the application, which is defined by the attribute `android:package` in the root `<manifest>` node. The package of our application is "com.ost.androidhelloworld," which we defined in the *New Project* wizard previously. Since Hermes activity is located in the root of our project (and not in a subfolder), `".HermesActivity"` is sufficient for the value of the `android:name` attribute. This is the only required attribute for activity nodes, but there are many other optional parameters; for example, `android:label` specifies the text that appears in our output. We'll cover a few more of these attributes in lessons to come, but feel free to explore the other possible attributes on your own.

Activity Class

In the MVC (Model-View-Controller) design pattern, the Activity class is considered the Controller. The MVC pattern is outside of the scope of this course, but if you are unfamiliar with it, I highly recommend that you take a few minutes to read about it in [Wikipedia](#). The Activity class is used to communicate with the View by populating it with data (from the Model) and handling or responding to user interactions with the View.

We'll look at the Activity class more later, but first we have to make one small change to our XML view. Open **activity_main.xml**. If you haven't made any changes to it since we worked on it before, the parent node will still be a `LinearLayout` with three child nodes: two `TextView`s and a `Button`. Modify the `Button` node in **activity_activity_main.xml** as shown:

/res/layout/activity_activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context=".MainActivity" >

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Hello World!" />

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="This is going to be the best app ever!" />

    <Button
        android:id="@+id/my_button"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Whoa, look, a button!" />

</LinearLayout>
```

Save the file, switch to the **MainActivity.java** file, then edit your code as shown:

CODE TO TYPE:

```
package com.ost.android1.helloworld;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;

public class MainActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

startHermes();
        Button myButton = (Button) findViewById(R.id.my_button);
        myButton.setOnClickListener(myButtonClickListener);
    }

    private OnClickListener myButtonClickListener = new OnClickListener() {
        @Override
        public void onClick(View v) {
            startHermes();
        }
    };

    public void startHermes() {
        Intent intent = new Intent(MainActivity.this, HermesActivity.class);
        startActivity(intent);
    }
}
```

After you've made those changes, when you fix the imports, you might be presented with multiple classes for `OnClickListener`. Be sure to choose to import the **View.OnClickListener** class.

Save your changes and run the project to see how these changes have affected the application. The home screen probably looks familiar to you, and now when you click the button it will actually do something! If everything is hooked up correctly in the code, the button will cause our `HermesActivity` to load, and the empty `Hermes` view should be visible.

So what did we do exactly? Let's review our changes, one by one:

OBSERVE:

```
Button myButton = (Button) findViewById(R.id.my_button);
```

First, we located and stored a reference to the `Button` into a **variable**. The `findViewById()` method comes from the parent **Activity** class, takes one Integer parameter, and returns a generic `View` object, so we must cast it to its specific class. Our parameter `R.id.my_button` is a resource reference to the id attribute we added to our XML view earlier. **R** is a class that is generated by the Android compiler and automatically populated with references to resources in the `/res` folder, because we referenced it earlier in the `onCreate()` method to load the view `R.layout.activity_main`.

OBSERVE:

```
myButton.setOnClickListener(myButtonClickListener);
```

Next, we attached a click listener to the button using the `setOnClickListener()` method, which takes a parameter of the **View.OnClickListener** type.

OBSERVE:

```
private OnClickListener myButtonClickListener = new OnClickListener() {  
    @Override  
    public void onClick(View v) {  
        startHermes();  
    }  
};
```

Finally, we created the **myButtonClickListener** object that we passed to `setOnClickListener`. `View.OnClickListener` is an interface that has one method, **`onClick(View view)`**, to handle each click event. The `View` parameter sent to the **`onClick()`** method is a reference back to the `View` that dispatched the click event. Then, we call the method we defined earlier to launch the `Hermes` Activity.

This is the most basic way of responding to clicks on Buttons in Android. Originally, this was the only way to handle clicks; but as of Android version 1.6, there is another, more efficient way. Since our project is already targeting version 2.2 of Android, let's update our code to use this alternate method of handling clicks. Open **`activity_main.xml`** class again and make the following change:

/res/layout/activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:tools="http://schemas.android.com/tools"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:paddingBottom="@dimen/activity_vertical_margin"  
    android:paddingLeft="@dimen/activity_horizontal_margin"  
    android:paddingRight="@dimen/activity_horizontal_margin"  
    android:paddingTop="@dimen/activity_vertical_margin"  
    android:orientation="vertical"  
    tools:context=".MainActivity" >  
  
    <TextView  
        android:layout_width="fill_parent"  
        android:layout_height="wrap_content"  
        android:text="Hello World!" />  
  
    <TextView  
        android:layout_width="fill_parent"  
        android:layout_height="wrap_content"  
        android:text="This is going to be the best app ever!" />  
  
    <Button  
        android:id="@+id/my_button"  
        android:layout_width="fill_parent"  
        android:layout_height="wrap_content"  
        android:text="Whoa, look, a button!"  
        android:onClick="handleMyButtonClick" />  
  
</LinearLayout>
```

Now, modify **`MainActivity.java`** below as shown:

MainActivity.java

```
package com.ost.android1.helloworld;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class MainActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button myButton = (Button) findViewById(R.id.my_button);
        myButton.setOnClickListener(myButtonClickListener);
    }

    private OnClickListener myButtonClickListener = new OnClickListener() {
    @Override
    public void onClick(View v) {
        startHermes();
    }
};

    public void startHermes() {
        Intent intent = new Intent(MainActivity.this, HermesActivity.class);
        startActivity(intent);
    }

    public void handleMyButtonClick(View view) {
        startHermes();
    }
}
```

Run the project again to test the code. The app will function the same way it did before. This change simplifies the code for handling clicks significantly. Instead of finding the button in the View and attaching a listener object to it, we use the XML `android:onClick` attribute to reference a method in our activity. There is no compiler-time checking for this method name; the method is presumed to be present in the Activity that implements the View. If the method is not present (or if it is defined incorrectly) then the application will throw an error when a user clicks the button. Methods referenced from the `android:onClick` attribute must have a return type of `void` and receive one parameter of type `View`.

Using this abbreviated method eliminates the need to store a reference to the button on the View. It's still sometimes necessary to get a reference to components on a View though, so you'll want to know how to use the `findViewById()` method correctly.

Basic View Components: Layouts and Buttons

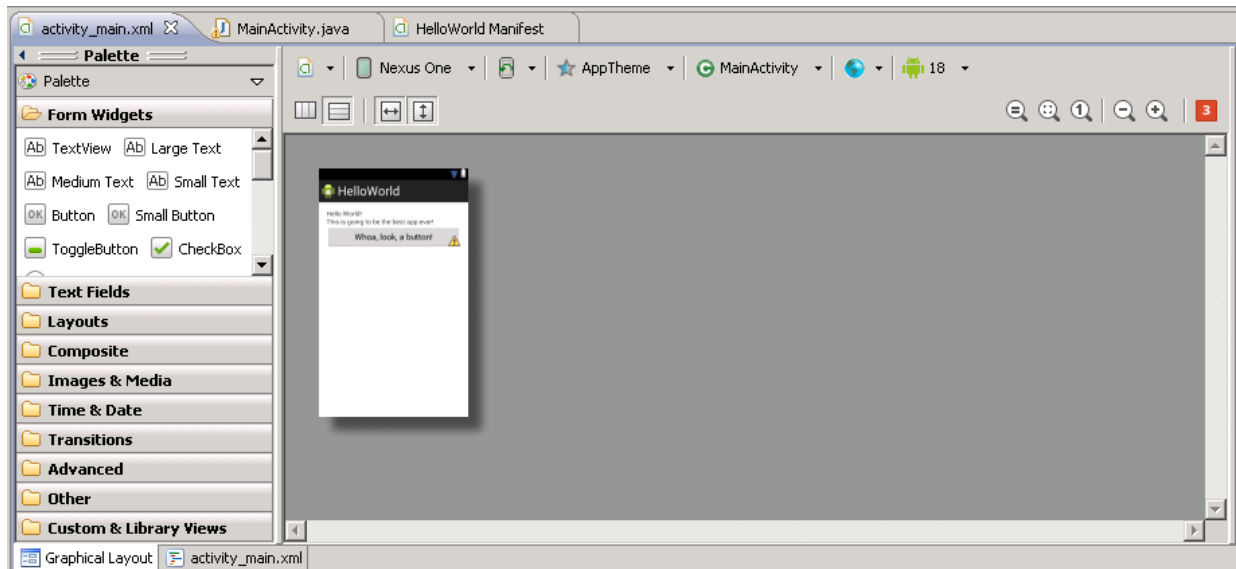
Layouts

Now that we know a bit more about controlling our Views, let's explore some more features of Android XML Views, starting with layouts. When you first create a View using the ADT wizard, it's populated with a `LinearLayout` tag for its root node automatically. The `LinearLayout` tag is used for arranging elements automatically, in a single direction, either horizontally or vertically. Horizontal layout is the default direction. To change the direction, use the `android:orientation` attribute.

There are two other common layouts to consider using when setting up your views, `RelativeLayout` and `FrameLayout`. `RelativeLayout` allows you to define position constraints for a view's components, relative to the parent and other components. `FrameLayout` puts each child on a separate layer (or frame), stacking them on top of each other.

View Components

There are many components available to use for Android views. We've already used two in our main view—TextView and Button. The standard view elements you would expect to see such as tabs, checkboxes, radio buttons, toggle buttons, and editable TextViews (called EditText), are already available. We're not going to cover each available component in this course, but you'll probably want to look into the available components on your own in the Android SDK on the [documentation site](#) or using the Graphical Layout XML editor provided by ADT.



Wrapping Up

We've covered a lot in this lesson! You should feel comfortable using the Activity class to find View components and handle click events, and you should know about many of the different types of View components available in Android. Feel free to experiment some more on your own until you feel confident in using those tools. See you in the next lesson, where we'll dig even further into Navigation and Data!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Navigation with Data

Welcome back! Earlier we mentioned starting a new Activity with an Intent. In this lesson, we'll go over Activities in more depth. We'll also talk more about the Intent class, and various methods for sharing data between activities.

Working with Intent

In the previous lesson we started a new Activity by creating an Intent and sending it to our current activity's `startActivity()` method. Here's the code we used in our MainActivity class to start the **HermesActivity**:

OBSERVE:

```
Intent intent = new Intent(MainActivity.this, HermesActivity.class);
startActivity(intent);
```

The Intent class in Android is used for much more than just starting Activities. Think of the Intent class as your way of letting the Android OS know of your "intent" to perform an action.

For example, to start a Service, you call **Activity.startService** and send an **intent** as the parameter. You can also use Intents to request that an action be performed in another application, such as opening a web page in the browser, sending a text message, or sending an email. Let's try doing that last one now.

First, we'll add a new button to our view to start this action. Edit **activity_main.xml** as shown here:

/res/layout/activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context=".MainActivity" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!" />

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="This is going to be the best app ever!" />

    <Button
        android:id="@+id/my_button"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Whoa, look, a button!"
        android:onClick="handleMyButtonClick" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Send Email"
        android:onClick="handleSendEmailClick" />

</LinearLayout>
```

Now, in **MainActivity.java**, add the logic to be performed when this button is clicked:

MainActivity.java

```
package com.ost.android1.helloworld;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class MainActivity extends Activity {

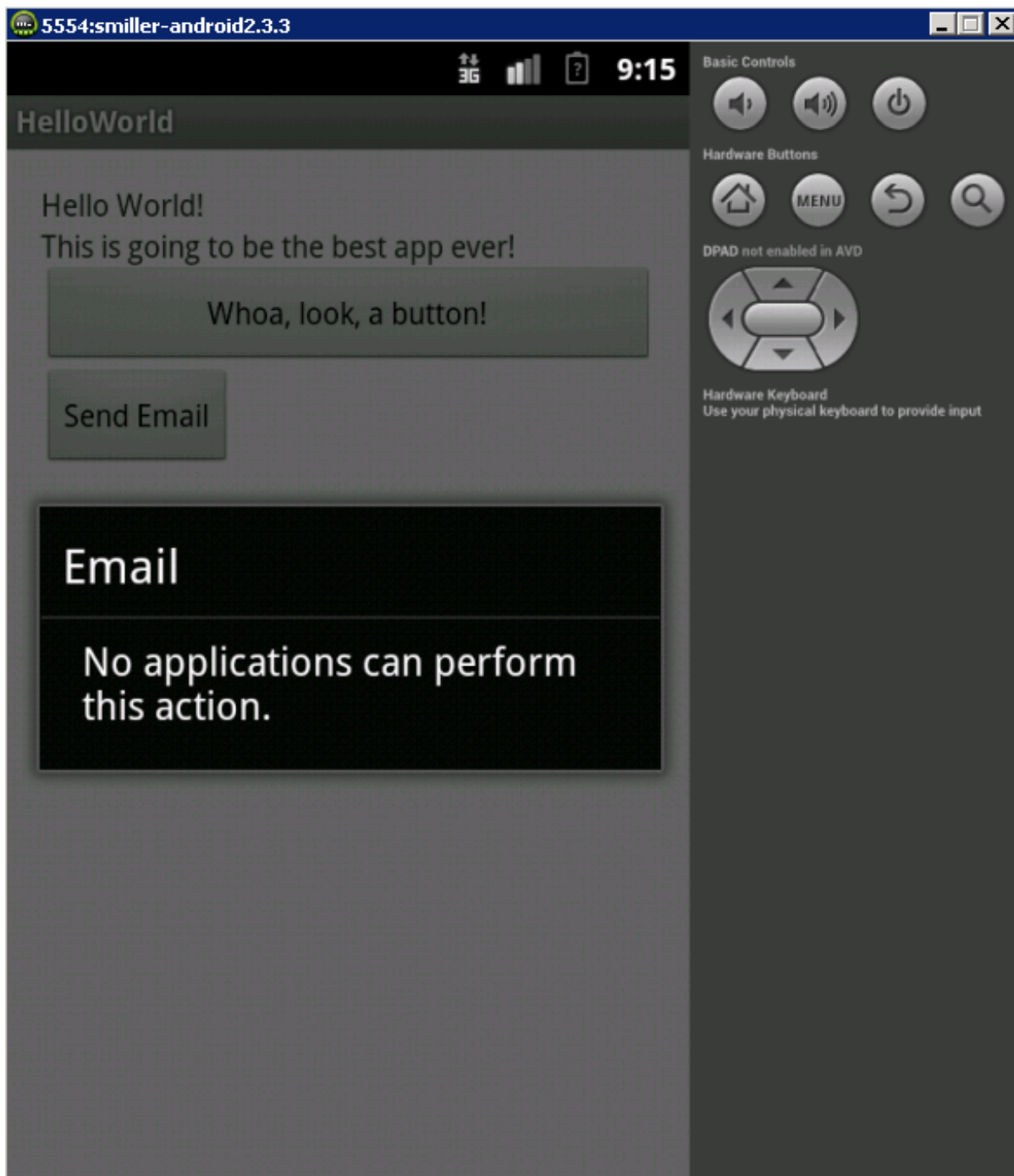
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void startHermes() {
        Intent intent = new Intent(MainActivity.this, HermesActivity.class);
        startActivity(intent);
    }

    public void handleMyButtonClick(View view) {
        startHermes();
    }

    public void handleSendEmailClick(View view) {
        Intent emailIntent = new Intent(Intent.ACTION_SEND);
        emailIntent.setType("plain/text");
        startActivity(Intent.createChooser(emailIntent, "Email"));
    }
}
```

Save and run it. If we try to test this now using the Android emulator, we won't see much of anything aside from a warning message.



This is because the default Android emulator doesn't come pre-bundled with any applications that support sending email. Specifically, the emulator doesn't have any applications that handle the `Intent.ACTION_SEND` action.

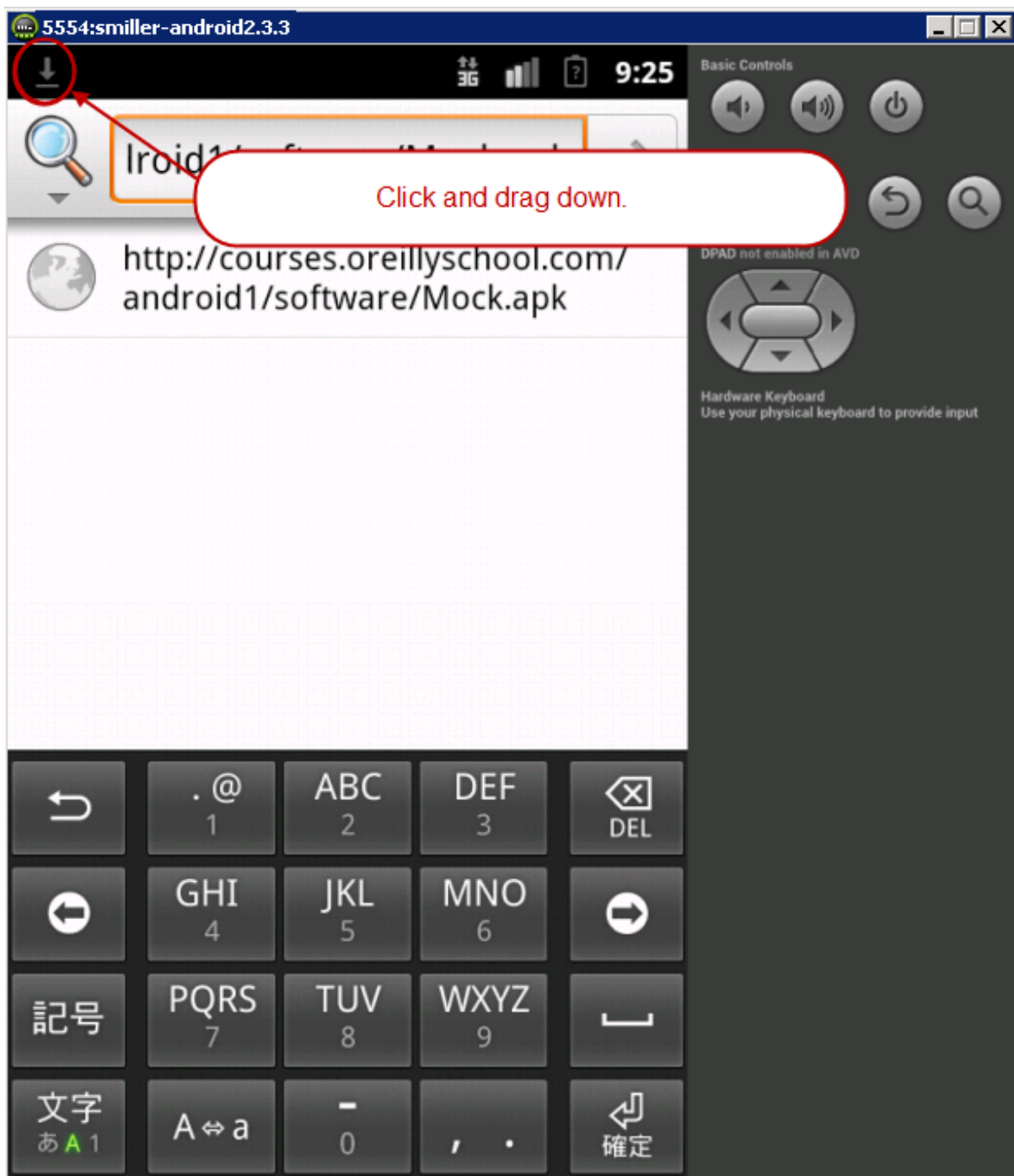
The best way to test this code would be to install the application on an actual device. But even if you own an Android device, you won't be able to install the application to it because the remote desktop connection environment can't recognize a device attached to your local computer. You would have to set up the android SDK and developer environment on your own computer in order to install to your own device. That's pretty extreme for our purposes, but don't worry—we can work around that.

An Emulator Email Alternative

I've created a basic mock application to handle the email intent, which you can download directly from the emulator. Open the browser on the emulator and type the url
<http://courses.oreillyschool.com/android1/software/Mock.apk>:

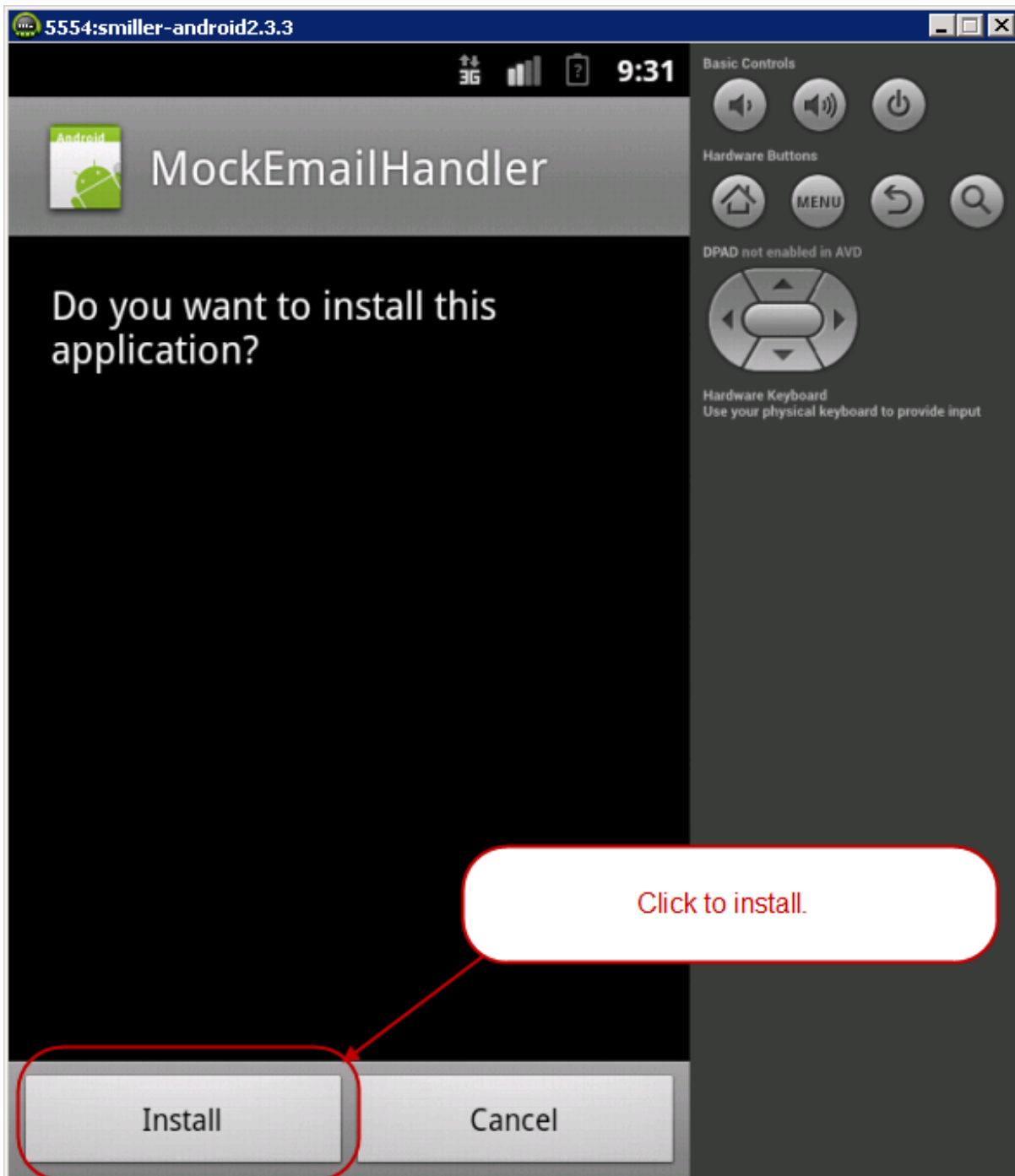


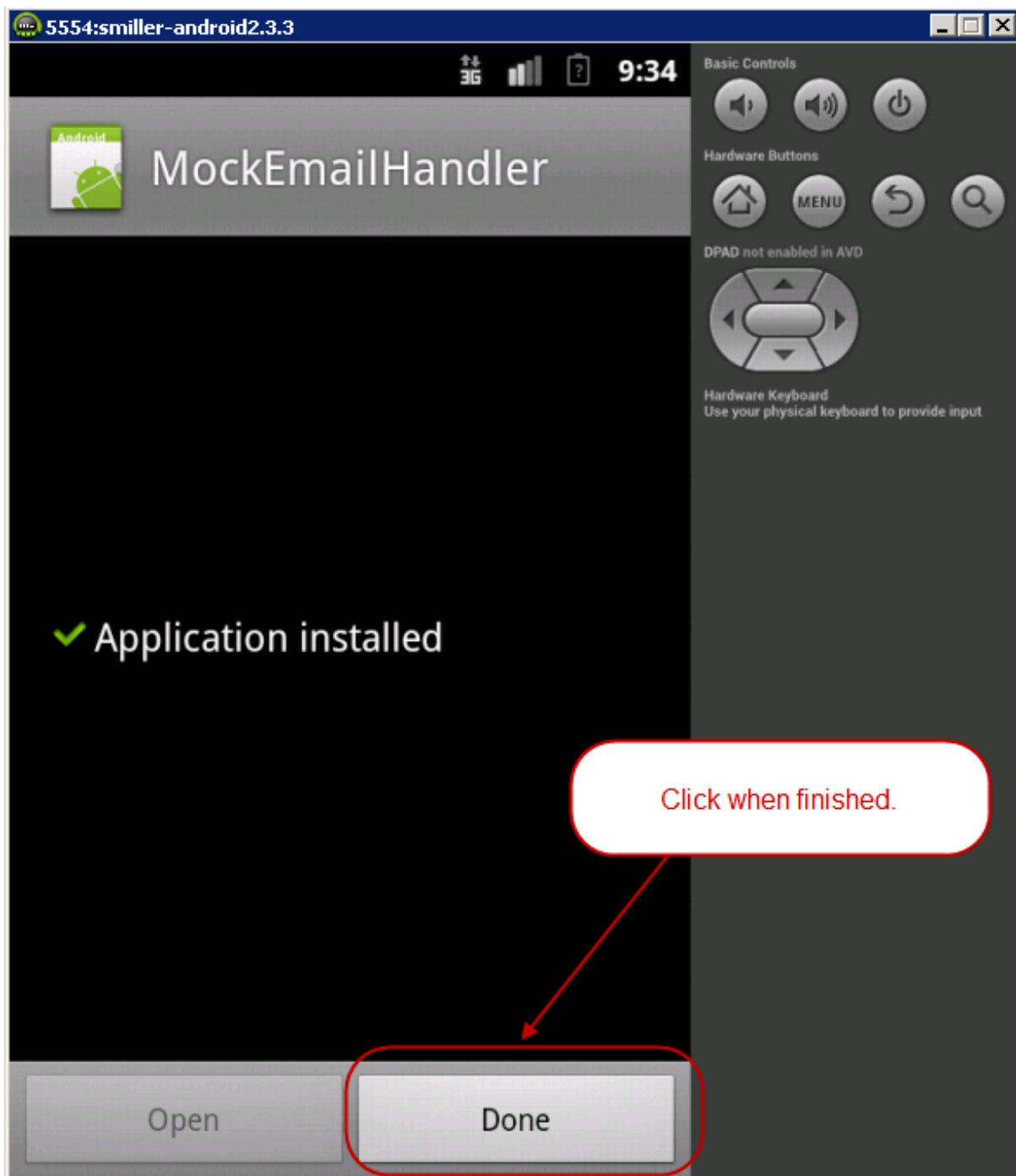
This downloads the application. Once it finishes downloading, drag down the window notification shade, and click on the download complete notification to install it:



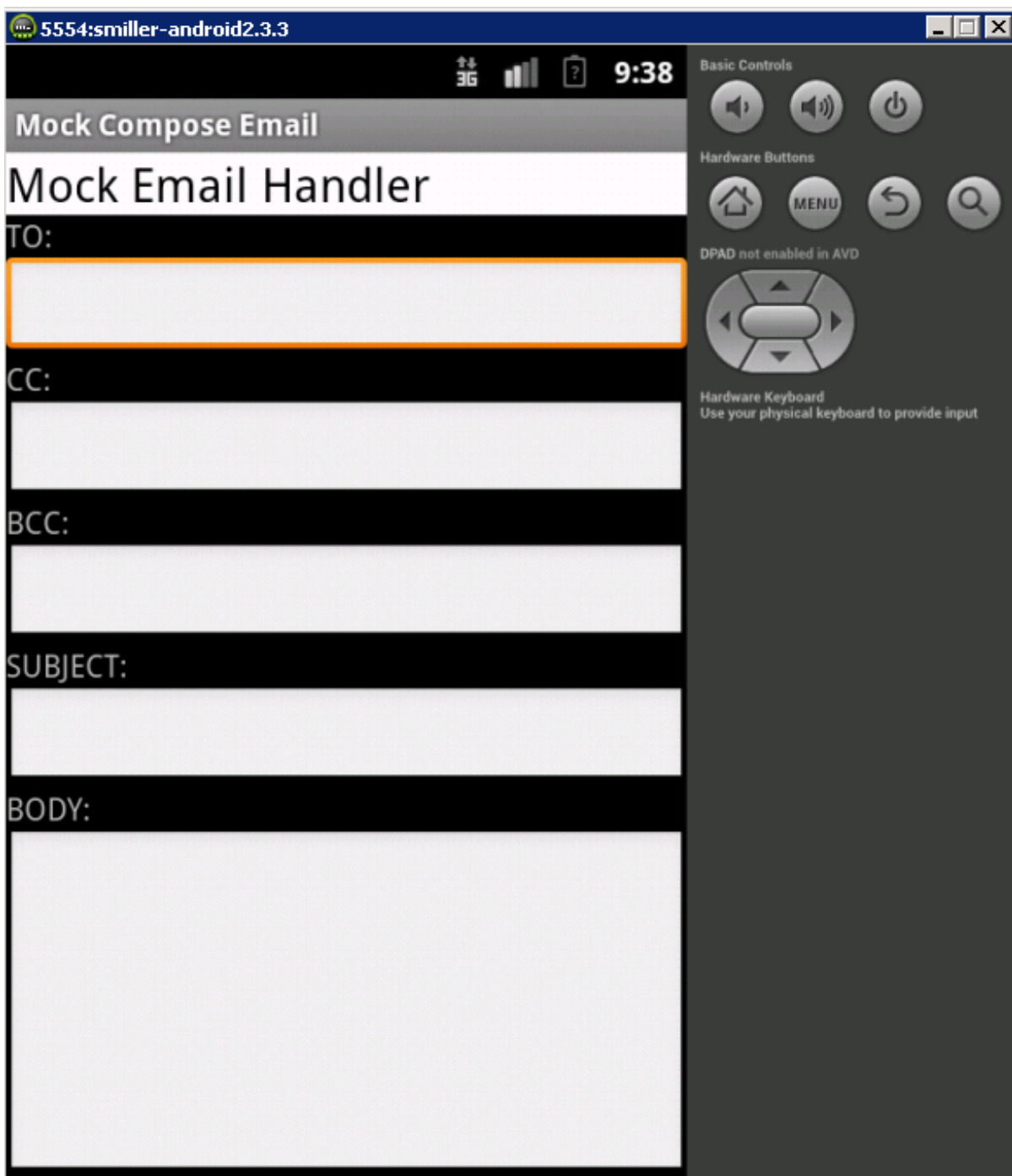
When the icon appears, click and drag anywhere in the top bar to pull down the "window shade."







This application won't actually send email, but now, when you test the code we wrote earlier, you'll see more than just the "No applications can perform this action" message. Instead, you'll see this:



Using this emulator, we can also define the fields of the email—such as the **To**, the **Subject**, and the **Body**—using the **Intent.putExtra** method:

OBSERVE:

```
public void handleSendEmailClick(View view) {
    Intent emailIntent = new Intent(Intent.ACTION_SEND);
    emailIntent.setType("plain/text");
    emailIntent.putExtra(android.content.Intent.EXTRA_EMAIL, new String[]{"predefined@mail"});
    emailIntent.putExtra(android.content.Intent.EXTRA_SUBJECT, "predefined Subject");
    startActivity(Intent.chooseIntent(emailIntent));
}
```

For a list of available Intent actions, see the [android documentation site for the Intent class](#).

Sharing Data Between Activities

Sometimes you'll need to pass data from one activity to another. We can do that using the Intent class as well. In fact, you've already done that once before in our example when you started the email Intent by using the **Intent.putExtra** method. Let's update our application to send some data back and forth between the MainActivity and the HermesActivity.

Sending Data to a New Activity

First, let's add an EditText to our **activity_main.xml** so we can get some user-defined text.

/res/layout/activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context=".MainActivity" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!" />

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="This is going to be the best app ever!" />

    <Button
        android:id="@+id/my_button"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Whoa, look, a button!"
        android:onClick="handleMyButtonClick" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Send Email"
        android:onClick="handleSendEmailClick" />

    <EditText
        android:id="@+id/my_edit_text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

</LinearLayout>
```

Next, edit **MainActivity.java** as shown:

MainActivity.java

```
package com.ost.android1.helloworld;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;

public class MainActivity extends Activity {

    private EditText myEditText;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        myEditText = (EditText) findViewById(R.id.my_edit_text);
    }

    public void startHermes() {
        Intent intent = new Intent(MainActivity.this, HermesActivity.class);
        intent.putExtra(HermesActivity.MY_EXTRA, myEditText.getText().toString());
    };

    startActivity(intent);
    }

    public void handleMyButtonClick(View view) {
        startHermes();
    }

    public void handleSendEmailClick(View view) {
        Intent emailIntent = new Intent(Intent.ACTION_SEND);
        emailIntent.setType("plain/text");
        startActivity(Intent.createChooser(emailIntent, "Email"));
    }
}
```

intent.putExtra uses a key/value pair system to store and retrieve the data being shared. The first parameter is the key, and is always a String value. Because this value must be exactly the same for storing and retrieving the value from the Intent, it's a good idea to use a static constant value here that both Activities can access. We're using the value on HermesActivity, which we just defined.

The second parameter to Intent.putExtra is the value. This parameter must be a primitive data type (such as Integer, Long, Float, or String) or it must be an object that implements the Parcelable interface. For now, we're only going to be sharing primitives between our activities; we'll talk about using the Parcelable interface in a later lesson.

When you save this file, you'll see a compiler error on the second line of the startHermes method. This is because we haven't defined the MY_EXTRA variable in HermesActivity yet. Let's do that before we proceed any further. Update **HermesActivity.java** as shown:

HermesActivity.java

```
package com.ost.android1.helloworld;

import android.app.Activity;

public class HermesActivity extends Activity {

    public static final String MY_EXTRA = "myExtra";
}
```

Save your changes and run the project; you'll see the new EditText field on the screen.

Returning Data to the Previous Activity

When sending data to a previous Activity, we use the Intent class, but the process is a bit different from the process used when sharing in the other direction. First of all, if an Activity expects to receive data from an Activity it starts, then it needs to use a different method to start that Activity.

We also need to add another method to handle receiving the data. Make the changes to **MainActivity.java** as shown:

MainActivity.java

```
package com.ost.android1.helloworld;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;

public class MainActivity extends Activity {

    private EditText myEditText;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        myEditText = (EditText) findViewById(R.id.my_edit_text);
    }

    public void startHermes() {
        Intent intent = new Intent(MainActivity.this, HermesActivity.class);
        intent.putExtra(HermesActivity.MY_EXTRA, myEditText.getText().toString());
startActivity(intent);
        startActivityForResult(intent, HermesActivity.EXTRA_REQUEST);
    }

    public void handleMyButtonClick(View view) {
        startHermes();
    }

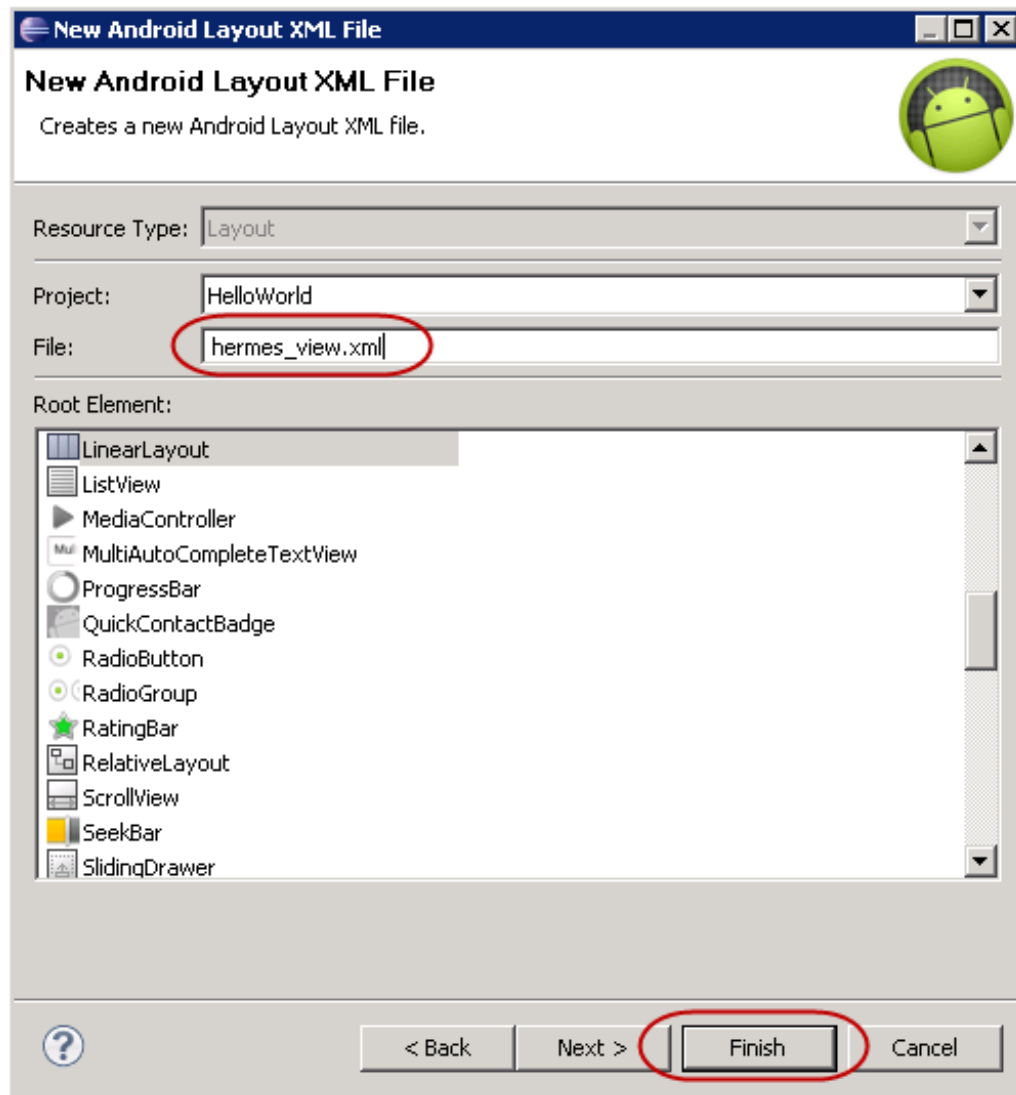
    public void handleSendEmailClick(View view) {
        Intent emailIntent = new Intent(Intent.ACTION_SEND);
        emailIntent.setType("plain/text");
        startActivity(Intent.createChooser(emailIntent, "Email"));
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        super.onActivityResult(requestCode, resultCode, data);

        switch(requestCode) {
            case HermesActivity.EXTRA_REQUEST:
                if (resultCode == RESULT_OK) {
                    String stringExtra = data.getStringExtra(HermesActivity.MY_EXTRA);
                    myEditText.setText(stringExtra);
                }
                break;
        }
    }
}
```

Next, we'll need to update both HermesActivity and its view. Well, actually, we haven't created a view for

HermesActivity yet, so let's start there. To create the new layout XML file, we'll use the ADT wizard. Select **File | New | Other** and choose **Android XML Layout File** in the **Android** folder. Name the file **hermes_view.xml** and click **Finish**. Leave the rest of the settings at their default values.



Click the **hermes_view.xml** tab at the bottom and make these changes to the file:

CODE TO TYPE:

```
<?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="match_parent"
    android:orientation="vertical">

    <EditText
        android:id="@+id/hermes_edit_text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
    />

    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Finish"
        android:onClick="onFinishClick"
    />

</LinearLayout>
```

Finally, we'll update **HermesActivity** to handle the data passed to it from the previous Activity, apply that data to the EditText, and respond to the **Finish** button being clicked by closing the Activity and sending the data back to the former Activity. Modify your code as shown:

CODE TO TYPE:

```
package com.ost.android1.helloworld;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;

public class HermesActivity extends Activity {

    public static final String MY_EXTRA = "myExtra";

    public static final int EXTRA_REQUEST = 0;

    private EditText hermesEditText;

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

        hermesEditText = (EditText) findViewById(R.id.hermes_edit_text);
        Intent i = getIntent();
        if (i.hasExtra(MY_EXTRA))
            hermesEditText.setText(i.getStringExtra(MY_EXTRA));
    }

    public void onFinishClick(View view) {
        String text = hermesEditText.getText().toString();
        Intent i = new Intent();
        i.putExtra(MY_EXTRA, text);
        setResult(RESULT_OK, i);
        finish();
    }
}
```

Run the application and test the code. The processes for sending data to an Activity and receiving data back from a started Activity are really similar. Both involve storing and retrieving data by use of an Intent object. You'll be able to make changes to the EditText in either activity, and then see the result when navigating to the other Activity using the **Whoa, look, a Button!** and **Finish** Buttons.

Application Class

You can also share data between multiple Activities throughout an Application using a custom Application class. As we mentioned earlier, every App on Android has a single *Application* class. This class is essentially a singleton (a design pattern that restricts the instantiation of a class to one object), and we can override the class with our own custom extension of the Application class to store state data.

Note

Do not abuse the singleton model in the Application class. The Android developer documentation on developer.android.com recommends using the Application class only for storing session state. You could also just store your state data on a helper class using public static variables. This would allow you to keep your code more modular and remove any dependencies on the Application framework.

Let's make a custom Application class now to store some application data. First, create a new class called **MyApplication** and make it extend the **android.app.Application** class.

New Java Class

Java Class
Create a new Java class.

Source folder: HelloWorld/src Browse...

Package: com.ost.android1.helloworld Browse...

☐ Enclosing type: Browse...

Name: MyApplication

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass: android.app.Application Browse...

Interfaces: Add... Remove

Which method stubs would you like to create?
☐ public static void main(String[] args)
☐ Constructors from superclass
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))
☐ Generate comments

Finish Cancel

The Application class has lifecycle methods similar to the Activity class. Any default data initialization should occur during the Application.onCreate() method. Edit your new class as shown:

```
MyApplication.java
package com.ost.android1.helloworld;

import android.app.Application;

public class MyApplication extends Application {

    public String defaultString;

    @Override
    public void onCreate() {
        super.onCreate();

        defaultString = "some default text";
    }

}
```

To get the Application to use our new class, we'll need to update the **AndroidManifest.xml** file. Update the **<application>** tag with a reference to the new class:

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.ost.android1.helloworld"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="10"
        android:targetSdkVersion="10"/>

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:name="MyApplication" >
        <activity
            android:label="@string/app_name"
            android:name=".MainActivity" >
            <intent-filter >
                <action android:name="android.intent.action.MAIN" />

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

        <activity android:name=".HermesActivity"
            android:label="Hermes Activity"/>

    </application>

</manifest>
```

Now that we've hooked up our new class properly, we just need to get a reference to it from our activities. Add the following to MainActivity.java:

CODE TO TYPE:

```
package com.ost.android1.helloworld;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;

public class MainActivity extends Activity {

    private EditText myEditText;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        myEditText = (EditText) findViewById(R.id.my_edit_text);

        MyApplication app = (MyApplication) getApplication();
        myEditText.setHint(app.defaultString);
    }

    public void startHermes() {
        Intent intent = new Intent(MainActivity.this, HermesActivity.class);
        intent.putExtra(HermesActivity.MY_EXTRA, myEditText.getText().toString());
        startActivityForResult(intent, HermesActivity.EXTRA_REQUEST);
    }

    public void handleMyButtonClick(View view) {
        startHermes();
    }

    public void handleSendEmailClick(View view) {
        Intent emailIntent = new Intent(Intent.ACTION_SEND);
        emailIntent.setType("plain/text");
        startActivity(Intent.createChooser(emailIntent, "Email"));
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        super.onActivityResult(requestCode, resultCode, data);

        switch(requestCode) {
            case HermesActivity.EXTRA_REQUEST:
                if (resultCode == RESULT_OK) {
                    String stringExtra = data.getStringExtra(HermesActivity.MY_EXTRA);
                    myEditText.setText(stringExtra);
                }
                break;
        }
    }
}
```

Test the application again. The default text for the EditText now contains the text we defined in our Application class ("some default text"). There are certainly better (and easier) ways of defining default text for a view component, but this will work just fine for our purposes right now.

Wrapping Up

Hopefully by now you're feeling comfortable with the Intent class and sharing data throughout your application. In the next lesson, we'll get to know the contents of the Android resources folder even better! See you there!



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Android Resources

Welcome back! Now that we've covered the basics of Navigation and sharing data, it's time to go further into the Android resources folder. In this lesson we'll cover string resources and how to use them in your code and views. Let's get started, shall we?

String Resources

If you've explored the `/res` folder, you may have noticed the `/res/values/strings.xml` file. This file defines and collects immutable string values for use in an application. Keeping all of your permanent strings defined in this file can be useful for solo developers or development teams—all Strings can be found, modified, and reused in a single location without having to hunt through every class just to find something like a typo, for example.

You've seen the strings that are already defined in `strings.xml` by default. Open `strings.xml` now (select the **strings.xml** tab at the bottom to edit it in xml mode) and add a few more strings:

`/res/values/strings.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">HelloWorld</string>
    <string name="action_settings">Settings</string>
    <string name="hello_world">Hello world!</string>
    <string name="header_text">Headers are cool.</string>
    <string name="subheader_text">Make sure you \"escape\" special characters like quot
es &amp; ampersands.</string>
    <string name="next">Go to Next Activity</string>
    <string name="send_email">Send Email</string>
    <string name="hint_text">This is hint text</string>
</resources>
```

Save the file.

Loading Strings in XML

That was pretty straightforward, but there's nothing to look at until we implement it in our view. Let's use our strings to populate the labels for our Buttons in our main view. Open `activity_main.xml` and update the views as shown:

/res/layout/activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context=".MainActivity" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        android:text="@string/header_text" />

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="This is going to be the best app ever!"
        android:text="@string/subheader_text" />

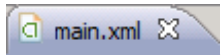
    <Button
        android:id="@+id/my_button"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Whoa, look, a button!"
        android:onClick="handleMyButtonClick" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Send Email"
        android:text="@string/send_email"
        android:onClick="handleSendEmailClick" />

    <EditText
        android:id="@+id/my_edit_text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/hint_text" />

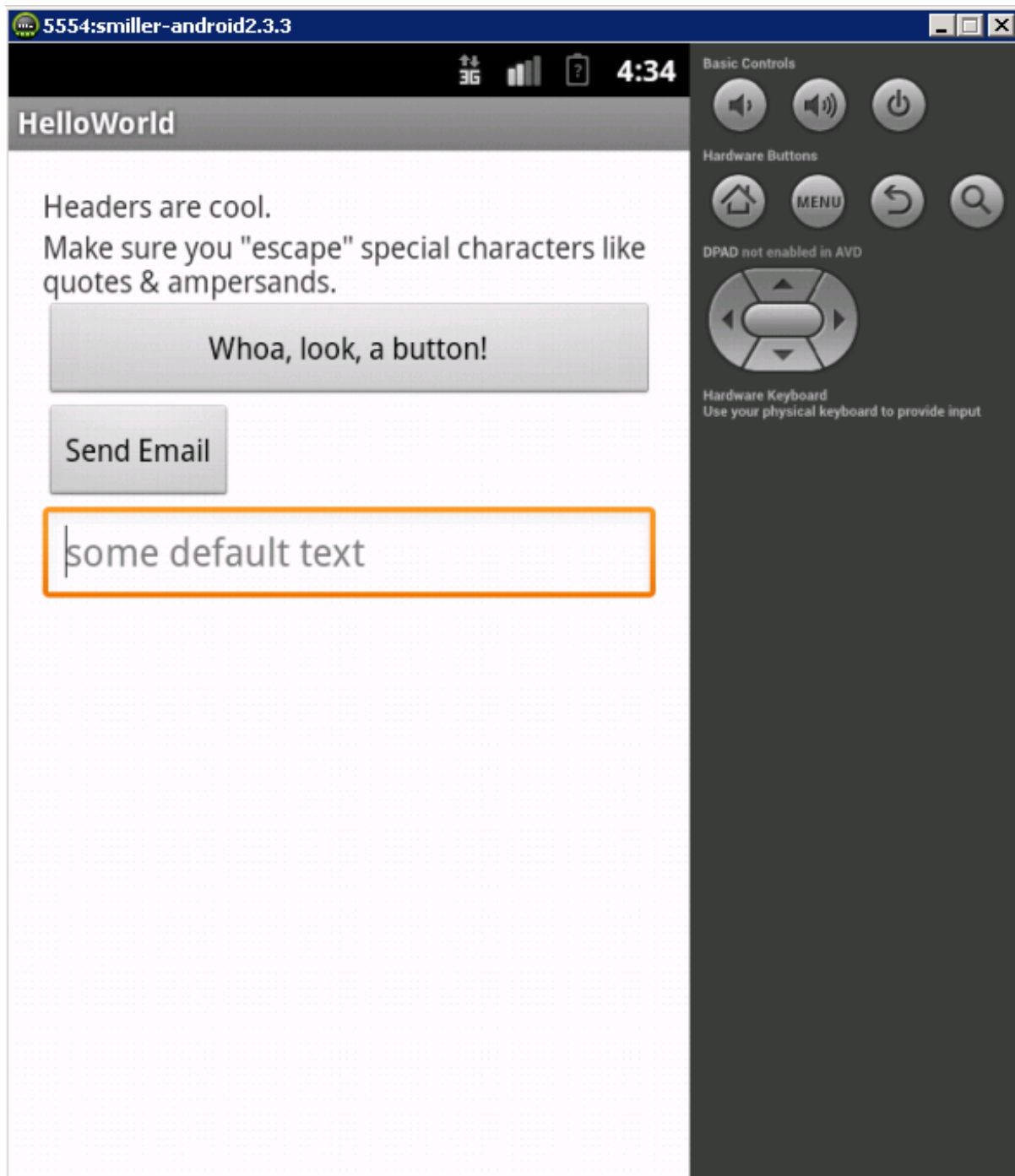
</LinearLayout>
```

You can see in your code that when referencing string resources in XML views, you use the format `@string/<string name>`. The "code complete" feature (**Ctrl+space**) also works in these views to help you find available resources and prevent typos. If you have trouble getting "code complete" to work in the editor, make sure the XML file has been opened in the appropriate editor. The file icon in the editor tab should look like this

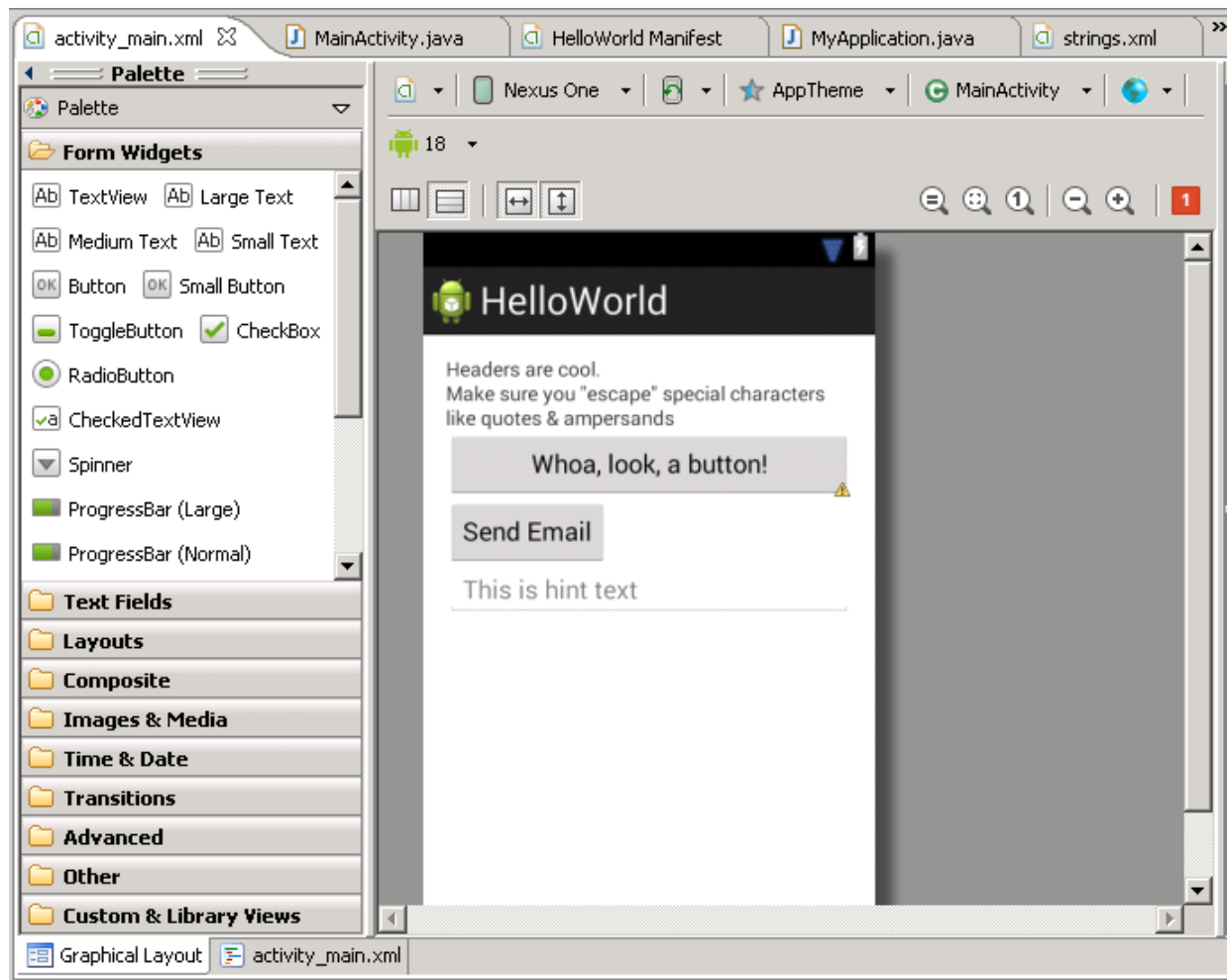


If you're seeing a different icon, close the file, and then reopen it by right-clicking the file name in the Package Explorer and selecting **Open With | Android Layout Editor**. That way you'll be sure that the "code complete" feature for resource values is working in your xml.

Save and run the application to test the results. Your first screen of the application will look like this:



At this point we don't actually have to run this in the emulator to make sure our view is correct. We can use the *Design* view of the Android Layout Editor, which is much faster. It can take a moment to initialize the first time the Design view is loaded for an Eclipse session, but ultimately it will save you valuable time.



The Design view won't always be able to render a pixel-perfect representation of the way a view will look in actual devices, but it should be sufficient for basic layouts and value testing like this.

Note

There's another nice little feature in some versions of ADT that can help you to create string resources. When you're working in your layout, you can select a string value that needs to be converted into a string resource, then use the **Refactor | Android | Extract Android String...** menu option to add the value to the strings file and update the component to use the new resource automatically. Incorporating this feature means you don't have to keep switching back and forth between your XML view layouts and the string resources.

Loading Strings in Code

Now that we've got our string resources loading in our views, let's use them in our code! I left out the *next* string resource intentionally so we could test that one in code; it could have been loaded in the XML like the others just as easily though. Open the **MainActivity.java** class and enter the code below into the onCreate method, as shown:

MainActivity.java

```
package com.ost.android1.helloworld;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;

public class MainActivity extends Activity {

    private Button myButton;
    private EditText myEditText;

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        myButton = (Button) findViewById(R.id.my_button);

        String next = getString(R.string.next);
        myButton.setText(next);

        myEditText = (EditText) findViewById(R.id.my_edit_text);

        MyApplication app = (MyApplication) getApplication();
        myEditText.setHint(app.defaultString);
    }

    private OnClickListener myButtonClickListener = new OnClickListener() {
        @Override
        public void onClick(View v) {
            startHermes();
        }
    };

    public void startHermes() {
        Intent intent = new Intent(MainActivity.this, HermesActivity.class);
        intent.putExtra(HermesActivity.MY_EXTRA, myEditText.getText().toString());
    };

    startActivityForResult(intent, HermesActivity.EXTRA_REQUEST);

    public void handleSendEmailClick(View view) {
        Intent emailIntent = new Intent(Intent.ACTION_SEND);
        emailIntent.setType("plain/text");
        startActivity(Intent.createChooser(emailIntent, "Email"));
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        super.onActivityResult(requestCode, resultCode, data);

        switch(requestCode) {
            case HermesActivity.EXTRA_REQUEST:
                if (resultCode == RESULT_OK) {
                    String stringExtra = data.getStringExtra(HermesActivity.MY_EXTRA);

                    myEditText.setText(stringExtra);
                }
                break;
        }
    }
}
```

Just like layout files (which are also located in the /res folder), string resources are loaded by using the generated R.java file. The actual string value is loaded by using the helper method `getString()`, which is defined on the encapsulated Context class. I haven't mentioned the Context class yet, but as we go further into the Android SDK, you'll see that Context is used frequently. You'll need a Context object to accomplish certain tasks (like loading resources or creating a database). Both Activity and Application classes encapsulate the Context class, so we typically pass one or the other as the Context.

Note

There is also a helper method available on the TextView component that takes the resource string id directly, which means the code could be simplified even more so that it's just a single line, for example: `myButton.setText(R.string.next)`.

The Resource Values Folder

In lessons to come, we'll create and use more files in the /res/values folder. The names of the files in the /res/values folder are chosen according to convention; the XML root node uses the `<resources>` tag. We used strings.xml file here to gather all the string definitions into a single file, but we could actually name the file whatever we want, just so long as the XML root node is the `<resources>` tag.

The files in the values folder are the only resource files where the name is not important. For every other file from other /res subfolders, the name is extremely important. This is because in those folders the name is essentially the 'id' value used to load the resource. For example, to access layouts (in code), we use `R.layout.<filename>`. The same pattern is used for every other subfolder in /res except values; the values subfolder adheres to this pattern: `R.<subfolder-name>.<filename>`. To load values defined in files in the values subfolder, we use the pattern `R.<value-type>.<name-attribute-value>`.

Note

Android restricts the names of files in the resources folder. Filenames can only contain lower-case characters a thru z, numbers 0 thru 9, and the underscore symbol. No capital letters, spaces, or special characters are allowed. The exception to this rule is for files in the res/values folder. For files in the /res/values folder, the rules apply to the values for the *name* attribute instead.

Wrapping Up

Using string resources in Android will help you to create better, more efficient code, and also make your code easier for other developers to read. Learn them, and love them! See you next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Drawables - Image Basics

In this lesson we'll work with resources again, but this time we'll focus on the Drawable class, which is used to manage images in Android. Let's get started!

Drawable Folders and Qualifiers

So far we've done all of our work on a single project, and it's starting to become a little cluttered. Let's close that project and start a new one for this lesson to clean up the workspace a bit and focus on just the current topics.

1. Right-click your **HelloWorld** root project folder and select **Close Project**. (Also, close any open files from the HelloWorld project in the Editor window.)
2. Create a new Android project named **Drawables**.
3. Set the package name of the project to **com.ost.android1.drawables**.
4. Uncheck the **Create custom launcher icon** box.
5. Add the project to the **Android1_Lessons** working set.

By default, images for Android applications should be stored in the **res/drawable** folder. There are already four different "drawable" folders in our project: **drawable-hdpi**, **drawable-ldpi**, **drawable-mdpi**, and **drawable-xhdpi**. The extended names for these folders are called "qualifiers." A qualifier is a string appended to one of the default folder names to indicate for which unique configuration that folder should be used.

The generated drawable folders in our application have qualifiers for the various Android phone screen resolution ranges. The "-hdpi" qualifier is for high resolution devices. This means a device that supports the high-density resolution range (~240dpi) for Android will attempt to load drawables out of the **drawable-hdpi** folder by default. "-mdpi" is for medium-density resolutions (~160dpi) and "-ldpi" is for low-density resolution devices (~120dpi).

Note

Qualifiers are used in Android for more than just screen resolution. They can be used to override any resource value for almost any hardware configuration, such as screen size, device layout, locale, and hardware support (such as a camera or trackball). We'll use qualifiers more in the coming lessons, but if you're curious to find out more about qualifiers now, check out this [article](#) on [Supporting Multiple Screens](#) on the Android developer documentation site.

Using Drawables

Our drawable folders are already populated with a single default image that is being used for the application icon. Now let's add another one to integrate into our application. To download the image, right-click on the image below and save the file to your **/res/drawable-hdpi** folder:



Note

The project folders are located on the V drive in the **/workspace/** folder; the full path where you should save the image is **V:\workspace\Drawables\res\drawable-hdpi**.

Now that we have the new image in place, let's get it loaded into our application. Open **activity_main.xml** from the **/res/layout/** folder and make these changes:

/res/layout/activity_main.xml

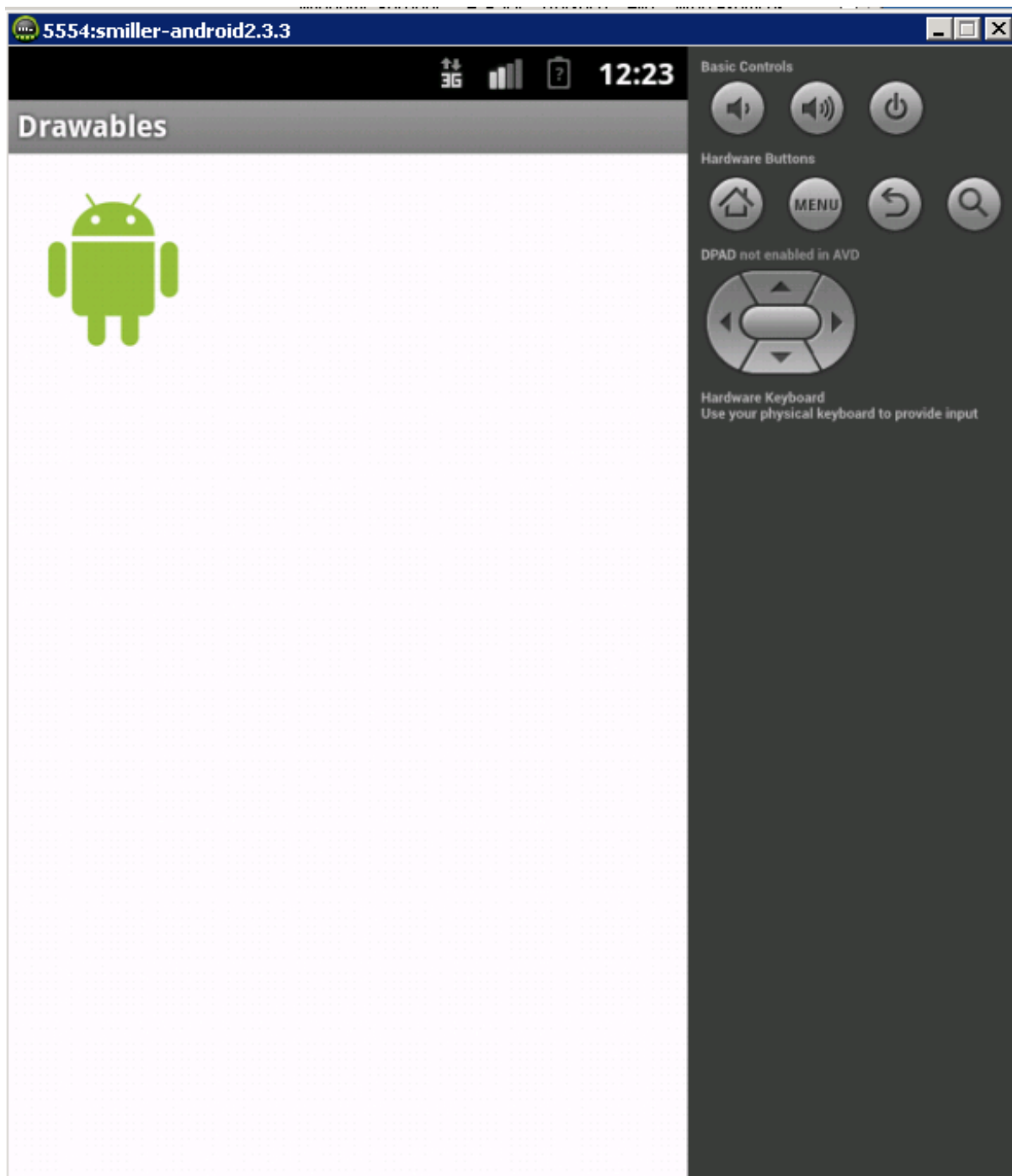
```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity" >

    <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/hello_world" />

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/ic_android_robot"
    />

</RelativeLayout>
```

Like all values in the resources folder, drawables are referenced using the @ symbol syntax. Let's run the application now to make sure that the ImageView loads the image correctly:



When you want to load a non-interactive image for display in your application, you'll typically use the **ImageView** component, like we just did. Other common use cases for loading images are for button icons and button skins.

Note

"Button skinning" is a little beyond the scope of this lesson, but we'll discuss how to implement a button skin later, when we cover Application skinning. Trust me, it will all make perfect sense to you later!

Let's add a button with the previous image as the button icon. Modify **activity_main.xml** as shown:

/res/layout/activity_main.xml

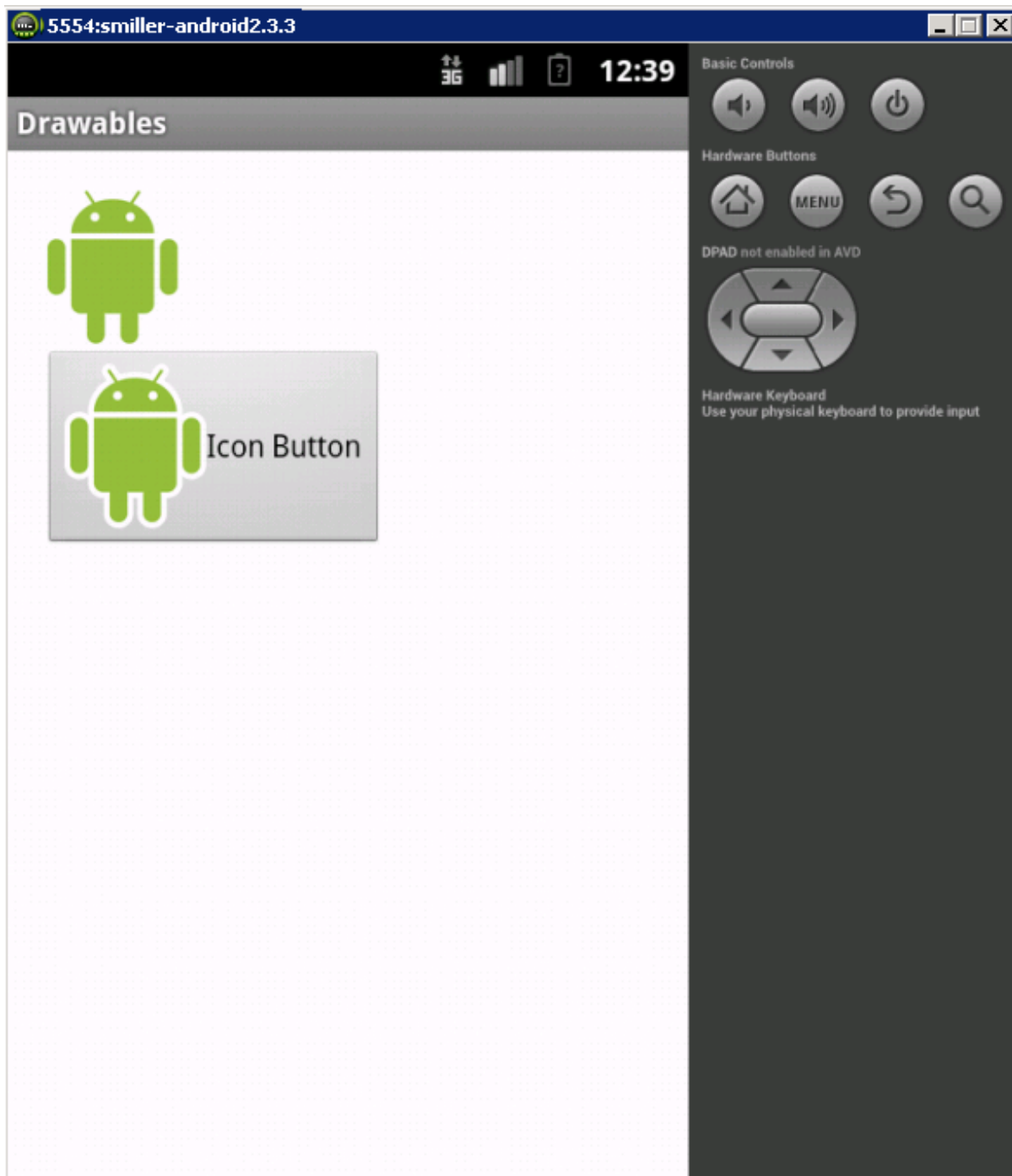
```
<RelativeLinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context=".MainActivity" >

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/ic_android_robot"
        />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:drawableLeft="@drawable/ic_android_robot"
        android:text="Icon Button"
        />

</RelativeLinearLayout>
```

Run the application again; your view will look like this:



The **drawableLeft** is a convenience property (alias) defined on the `TextView` class (of which `Button` is a subclass). As you may have guessed, there are additional properties called, **drawableTop**, **drawableRight**, and **drawableBottom**; they behave exactly as you'd expect.

Dimensions

So far we've defined the **layout_width** and **layout_height** attributes of our `Images` (and all of our components) as either **match_parent** or **wrap_content**. These are handy relative dimension properties, but when neither property is sufficient for your needs, you'll want to use a more specific dimension.

If you've ever developed a user interface for another application, you're probably used to defining your width and height dimensions in pixels. In Android, using precise pixels for dimensions is not recommended though, because Android devices come in so many different shapes and sizes, with so many different resolutions. This means that the number of available pixels on the screen can vary greatly by device. To address this issue, the Android SDK has its own unit for dimensions called "density independent pixels"—"dip" or "dp" for short. When you use "dip" units for your dimensions, the Android SDK will automatically scale the actual pixel dimension to an appropriate relative pixel size to keep the relative sizes and spacing the same for each device.

Let's update our view now to use the "dip" unit for some of our dimensions, so we can better control the size of our components. Make these changes to activity_main.xml:

/res/layout/activity_main.xml

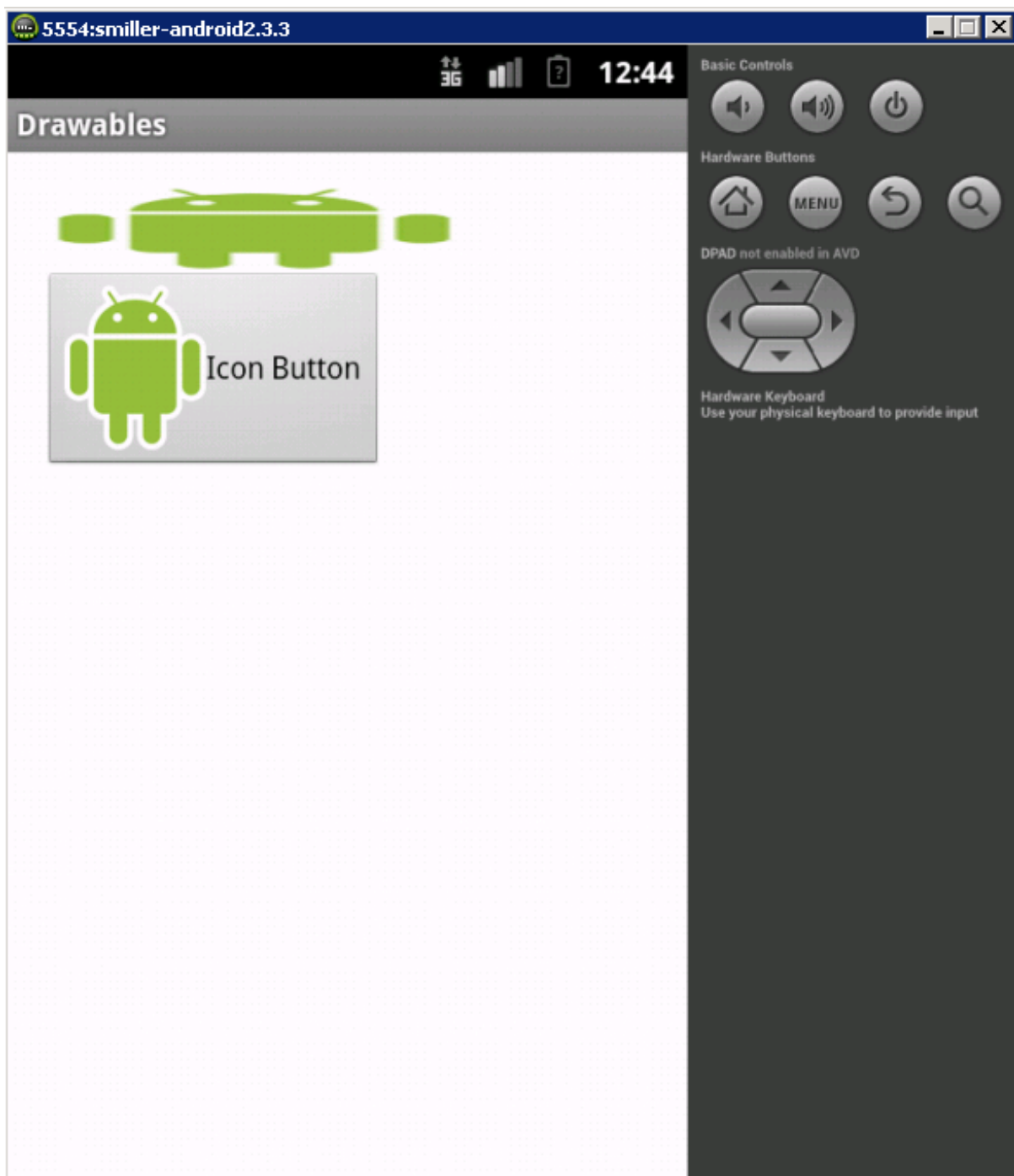
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context=".MainActivity" >

    <ImageView
        android:layout_width="wrap_content200dp"
        android:layout_height="wrap_content40dp"
        android:src="@drawable/ic_android_robot"
        android:scaleType="fitXY" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:drawableLeft="@drawable/ic_android_robot"
        android:text="Icon Button" />

</LinearLayout>
```

Save the file and give it a test run; the first screen of the application will look like this:



Note

There isn't any direct way to have a button resize its `drawableX` (`drawableLeft`, `drawableRight`, and so on) icons. You can change the dimensions of the button, but the image will remain its original size (and clip the edge of the button if the button is smaller than the image). Another way to approach this issue (without creating a new image) would be to use an XML drawable. We'll cover XML drawables in a future lesson.

In order to see a complete comparison of the differences between using density independent pixels and ordinary pixel units, you'd need to create a second emulator with slightly different dimensions and test the code on each device once using "dip" for your dimensions and again using "px." Doing that while using a remote desktop connection would be pretty time consuming though.

Note

When defining a font size for text components using exact pixels is not recommended either. Android provides an alternative unit called "scale independent pixels"—"sp" for short. These units behave exactly like "dip" units, but they will also be scaled by the user's default font scale if specified.

For an in-depth explanation of how dip units work, check out the [Android developer documentation site](#) regarding dimensions.

Image Padding

If you want to adjust the placement and spacing of the icon image inside your button, there are a few tricks you can use. First, there is a property called **drawablePadding**, which defines the minimum amount of padding the widget should use between the icon and the text content. This property adds padding only between the text and the icon, and only when there is a drawable icon defined as well. This will not create padding between the icon and the edge of the button. If you want to add space between the icon and the edge of the button, use the **padding** property just as you would for any layout component that has children.

Note

By default, the `drawableX` property will draw the icon as close as it can to the edge of the button. This is most noticeable when using a `drawableLeft` or `drawableRight` icon and the button has a `layout_width` value of "match_parent" so that it fills the entire width of the device. In that situation the icon will be drawn near the edge of the button and not near the text inside the button. So, in this instance the **drawablePadding** property will have no effect on the button.

Let's add some `drawablePadding` to the Button component in **activity_main.xml**:

/res/layout/activity_main.xml

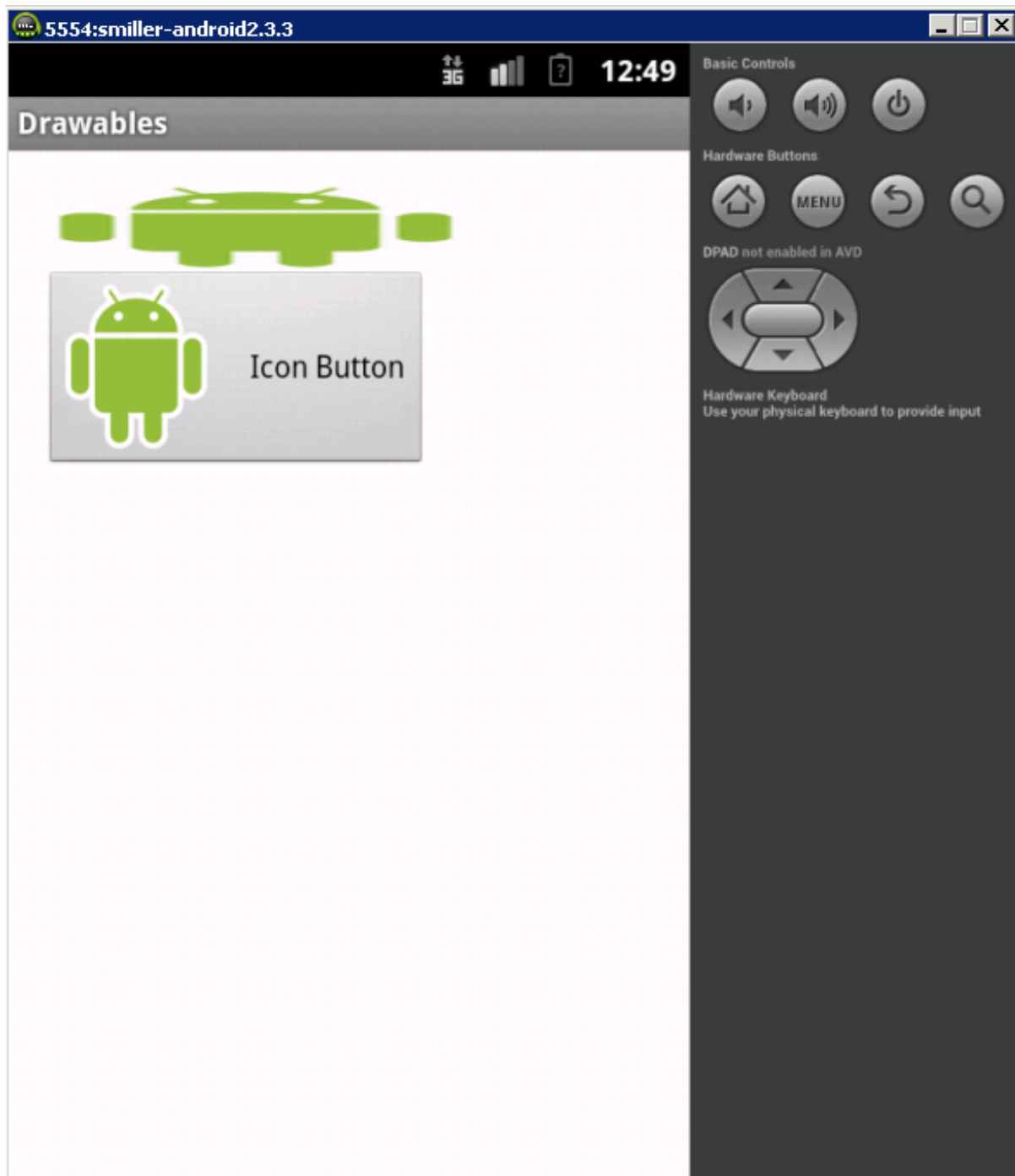
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context=".MainActivity" >

    <ImageView
        android:layout_width="200dp"
        android:layout_height="40dp"
        android:src="@drawable/ic_android_robot"
        android:scaleType="fitXY" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:drawableLeft="@drawable/ic_android_robot"
        android:drawablePadding="20dp"
        android:text="Icon Button" />

</LinearLayout>
```

Save and run it; you'll see something like this:



The only space that is affected is that between the icon and the text of the button. The space between the icon and the top, left, and bottom edges of the button remains unchanged.

The ImageButton Widget

If you have a button that needs only an icon (and no text on the button) then you should probably use an ImageButton. The ImageButton class is actually a subclass of ImageView (and not Button). There are actually very few differences between an ImageButton and an ImageView. Both have a **background** property that takes a drawable, as well as an **src** property that also takes a drawable, and both are clickable view components. However, by default, the ImageButton component will use the default skin for Button as its background drawable, while the ImageView does not have a default background.

Let's add an ImageButton component to our code:

/res/layout/activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context=".MainActivity" >

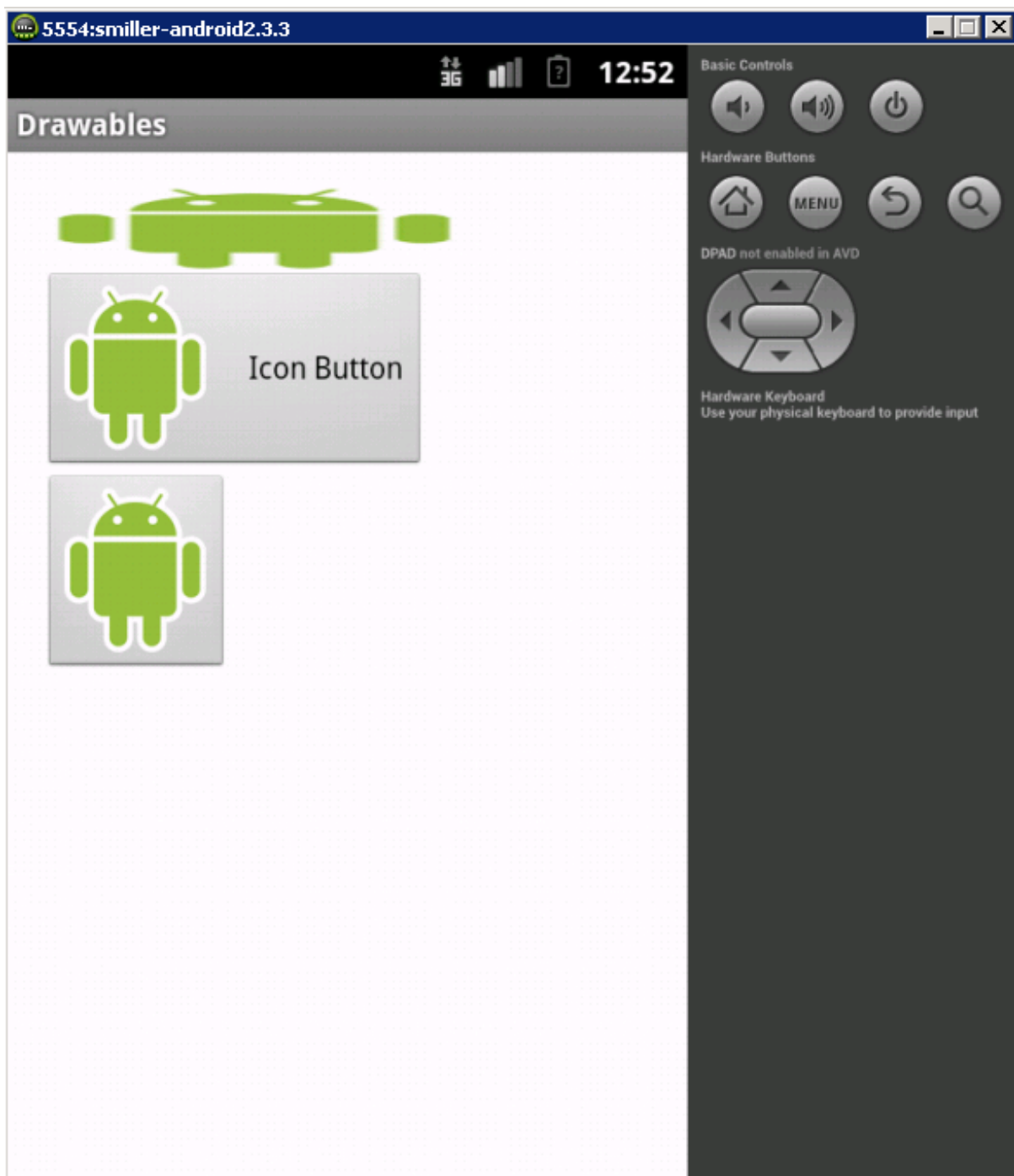
    <ImageView
        android:layout_width="200dp"
        android:layout_height="40dp"
        android:src="@drawable/ic_android_robot"
        android:scaleType="fitXY" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:drawableLeft="@drawable/ic_android_robot"
        android:drawablePadding="20dp"
        android:text="Icon Button" />

    <ImageButton
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/ic_android_robot" />

</LinearLayout>
```

Now run the app to verify the results:



As you can see, the ImageButton still has the default skin of the Button, with our icon drawn in the middle of the button. Since the ImageButton component is a subclass of ImageView and not Button, there is no **drawablePadding** attribute available (besides, it wouldn't be of any use for this component). It doesn't have a **text** attribute either, but it does have the **scaleType** property available to help you to define how the image is scaled inside of the view component.

Wrapping Up

We've covered a lot of important stuff in this lesson. I'm confident that you know how to use Images in your views, as well as how Android uses "density independent pixels" as a dimension unit.

We've spent a lot of time in the resources folder so far, and you have a pretty strong grasp of how to implement a majority of Android's view components. If you feel like you want to experiment a bit more with these concepts on your own, do it! In upcoming lessons, we'll get back to work in the Java classes. See you there!



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Lists

Good to see you again! This lesson will cover Android lists. Let's get started!

Implementing an Android List

Go ahead and start a new application. Select **File | New | Project | Android Application Project**, and create the application using these settings:

1. Name the Project **Lists**.
2. Type **com.oreillyschool.android1.lists** for the Package name.
3. Clear the **Create custom launcher icon** check box.
4. Add the project to the **Android1_Lessons** working set.

ListView

Before we can start implementing our list, we need to make one small change to our view; we need to add a list to it, of course! Open **activity_main.xml** in the **/res/layout/** folder and make these changes:

```
/res/layout/activity_main.xml

<RelativeLinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context=".MainActivity" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />

    <ListView
        android:id="@android:id/list"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        />

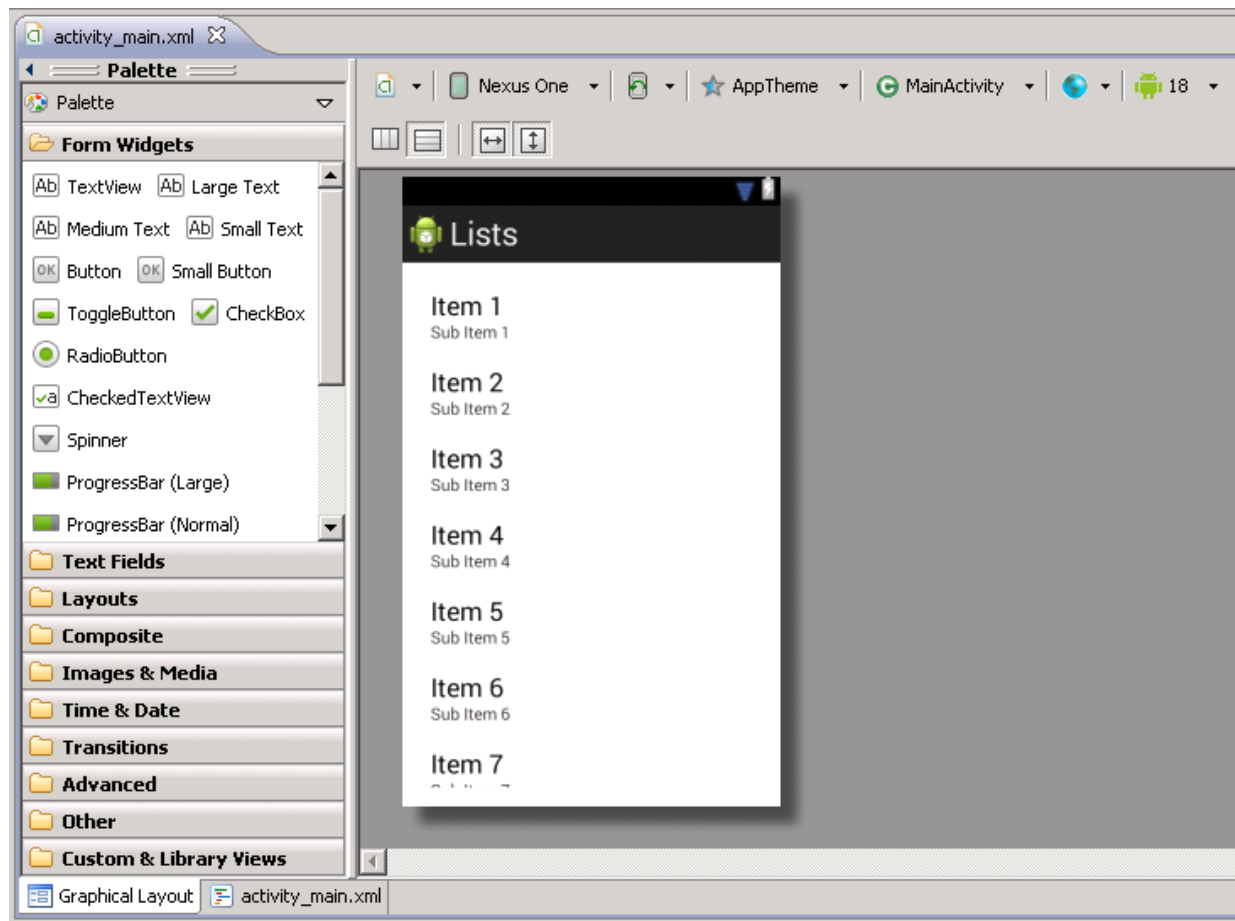
</RelativeLinearLayout>
```

Note

As of Android API 8, 2.2, the layout attribute **fill_parent** was replaced with the more accurate **match_parent** label.

The **ListView** component is the default List handler for Android. Notice that we used a slightly different "id" value from what we used before. (We'll discuss that in greater detail a bit later.)

There's not much to see in our program yet, but if you click the **Graphical Layout** tab in the XML editor, you will see a stubbed default List:



This view can show only fake stub data and will never reflect the actual content of your list. You'll need to run the application to test and make sure that your list items are working correctly. Before you can do that, though, you'll need to implement the list in code.

ListActivity

To use a List in your view, you'll want to use a different kind of Activity class called the ListActivity. ListActivity is a subclass of Activity, and while it is not required for implementing lists, it can make the setup and maintenance of a list much easier.

Open **MainActivity.java** and make these changes:

```
/src/MainActivity.java

package com.oreillyschool.android1.lists;

import android.app.ListActivity;
import android.os.Bundle;
import android.view.Menu;

public class MainActivity extends ListActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }
}
```

Remember earlier how we used a unique value for our "android:id" attribute in the XML view? This id value is actually predefined by the Android SDK. The code reference equivalent is **android.R.id.list**. However, unlike before, we don't need to find and store a reference to this view (using *findViewById*), because **ListActivity** has already taken care of that. **ListActivity** expects that the layout loaded by **setContentView** contains a list that contains that id value, loads that value, and manages it internally. Any interaction with the **ListView** component is then handled by helper methods available on the **ListActivity** class. If you wanted to subclass a regular **Activity** class, you would need to manage the **ListView** component manually.

Empty Lists

ListActivity also has the added benefit of showing an alternate view when its list is empty. Taking advantage of this feature requires another special Android id, **android.R.id.empty**. Let's open **activity_main.xml** again and add an "empty" view:

/res/layout/activity_main.xml

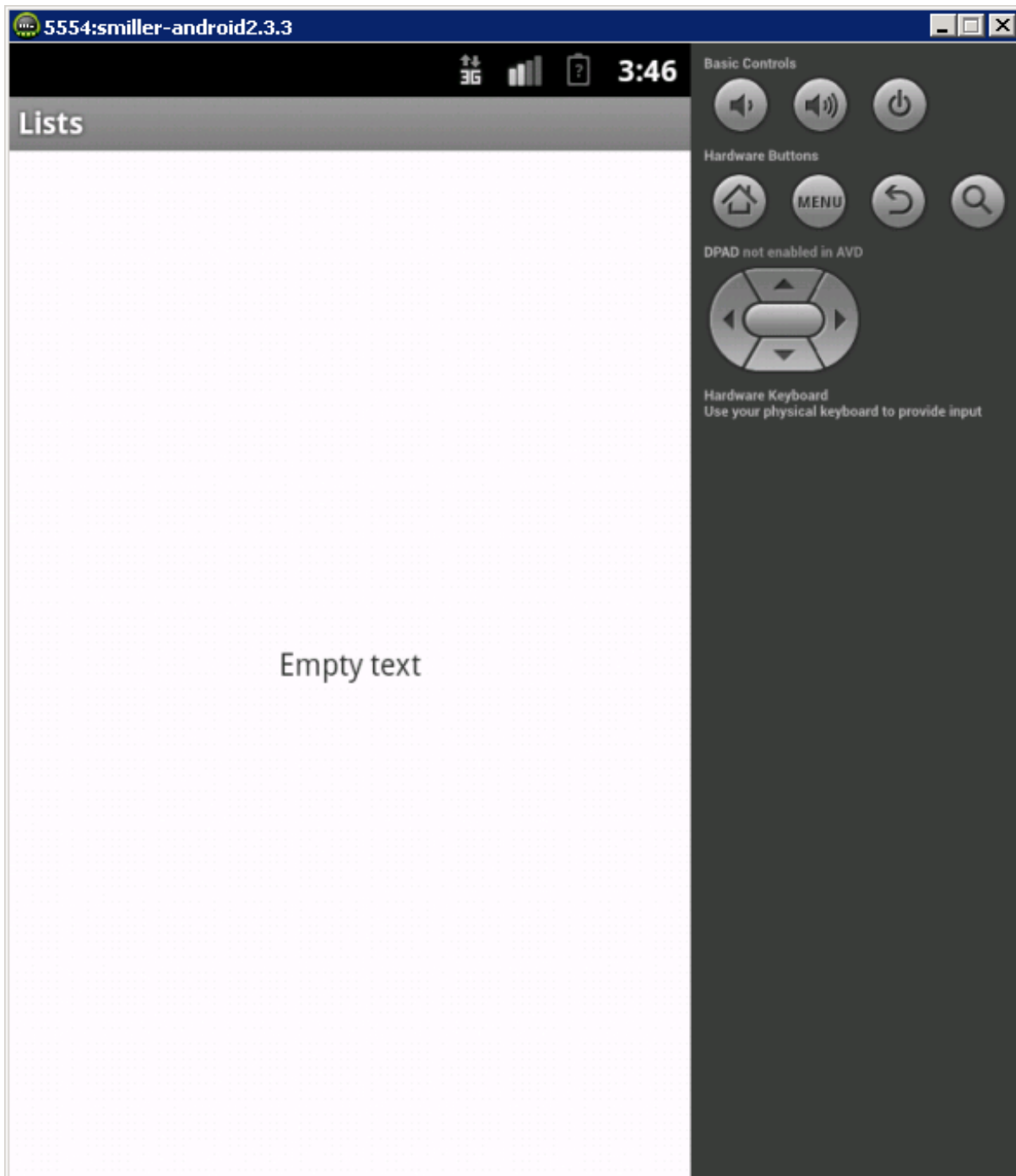
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context=".MainActivity" >

    <ListView
        android:id="@android:id/list"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        />

    <TextView
        android:id="@android:id/empty"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="@string/empty_text"
        android:gravity="center"
        />

</LinearLayout>
```

You'll also need to add a string named **empty_text** to the **strings.xml** file. You can do that manually or use the **Refactor | Android | Extract Android String...** shortcut we discussed earlier. Give the string whatever value you like. When you're finished, go ahead and run the application; you'll see that the empty text is shown, because our list doesn't have any data yet:



ListAdapter

A list isn't at all useful without data, so we'll need to work on that. To manage list data in Android we use an Adapter. A list view expects an object of the type *android.widget.ListAdapter*, which is an interface. Implementing the entire interface isn't necessary, though, because there are many default implementations available in the SDK that you can use that are less labor intensive. I personally prefer the **`android.widget.ArrayAdapter<T>`** class.

Let's get our ListView hooked up to an adapter with an instance of ArrayAdapter. Make these changes to **MainActivity.java**:

MainActivity.java

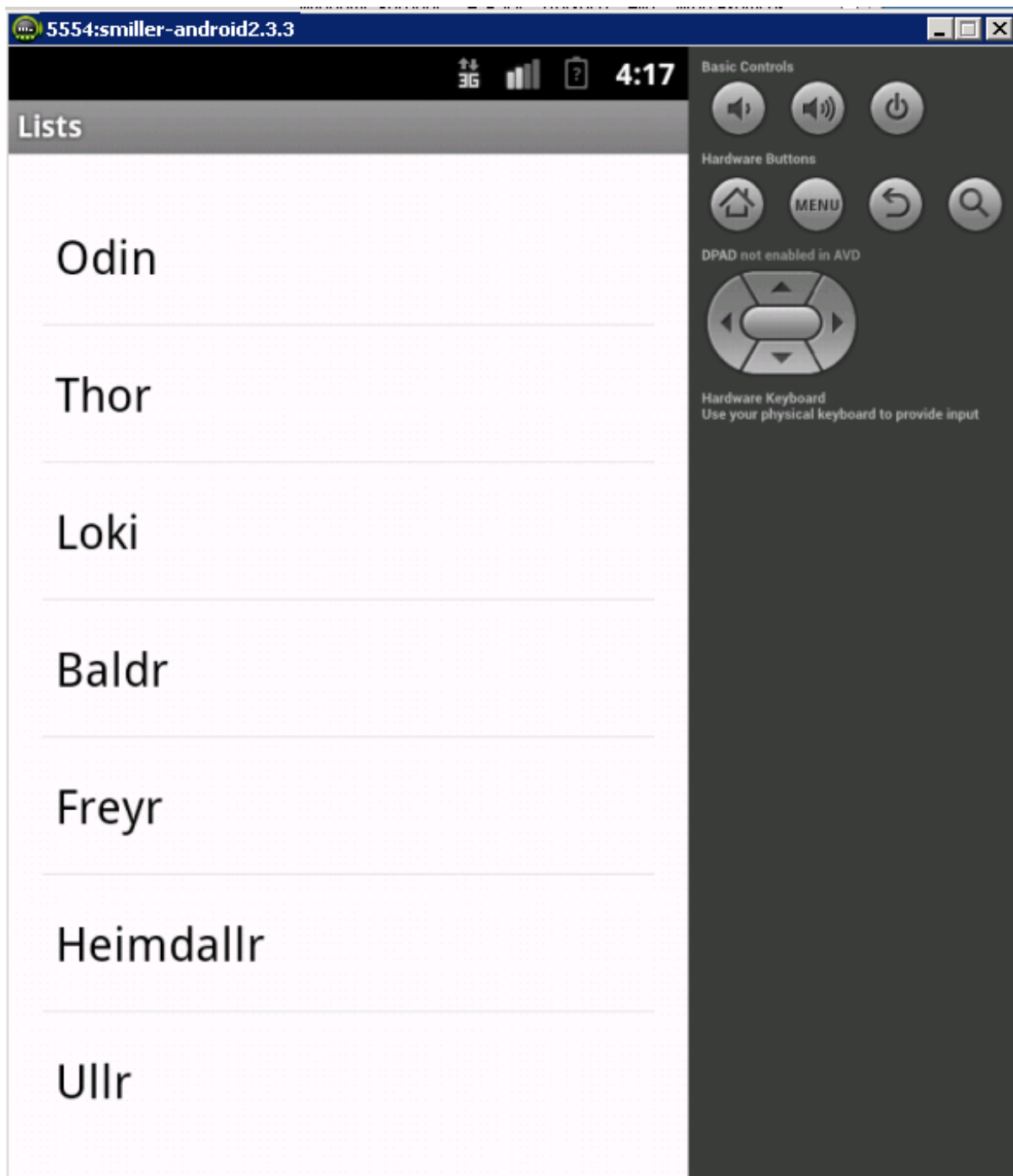
```
package com.oreillyschool.android1.lists;

import android.app.ListActivity;
import android.os.Bundle;
import android.widget.AdapterView;

public class MainActivity extends ListActivity {
    private String[] data = new String [] {
        "Odin",
        "Thor",
        "Loki",
        "Baldr",
        "Freyr",
        "Heimdallr",
        "Ullr",
        "Meili",
        "Hodr",
        "Forseti"
    };

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ArrayAdapter<String> adapter = new ArrayAdapter<String>(this, android.R.
        layout.simple_list_item_1, android.R.id.text1, data);
        setListAdapter(adapter);
    }
}
```

Now run the application and test your results. Your emulator should look like this:



Let's look at our code in some more detail and see exactly what's going on:

OBSERVE:

```
...

public class MainActivity extends ListActivity {
    private String[] data = new String [] {
        "Odin",
        "Thor",
        "Loki",
        ...
        "Forseti"
    };

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ArrayAdapter<String> adapter = new ArrayAdapter<String>(this, android.R.
layout.simple_list_item_1, android.R.id.text1, data);
        setListAdapter(adapter);
    }
}
```

First, we created some **basic stub data for the list**, otherwise the list would still be empty. Then we used **setListAdapter()**, a helper method on ListActivity that sets the adapter on the ListView component managed by the ListActivity.

Our constructor for ArrayAdapter has a signature that requires four parameters:

ArrayAdapter Constructor

```
ArrayAdapter(Context context, int resource, int textViewResourceId, T[] objects)
```

The first parameter is a **Context**. All Activity classes are subclasses of Context, so we just pass **this** as the value.

The next parameter, **int resource**, is a resource reference to the XML layout to be used for each item in the list. We used a basic layout that was already provided in the Android SDK: **android.R.layout.simple_list_item_1**. This layout contains only one component: a TextView.

The third parameter, **int textViewResourceId**, is more or less self-explanatory. It's an id reference to the TextView component that is contained in the previously defined resource's layout. The id reference to the TextView contained in the **simple_list_item_1** layout is **android.R.id.text1**.

The final parameter is to an Array of the data that will be used to populate the TextView component for each item in the list. Note that this Array is type-restricted to the bounded type parameter that was used for the constructor method, which in our case is **String**.

Sorting the Adapter

To sort the ArrayAdapter, just call **sort()** on the adapter and send it a *Comparator* object. Let's implement sort in our code now. Change **MainActivity.java** as shown:

CODE TO TYPE:

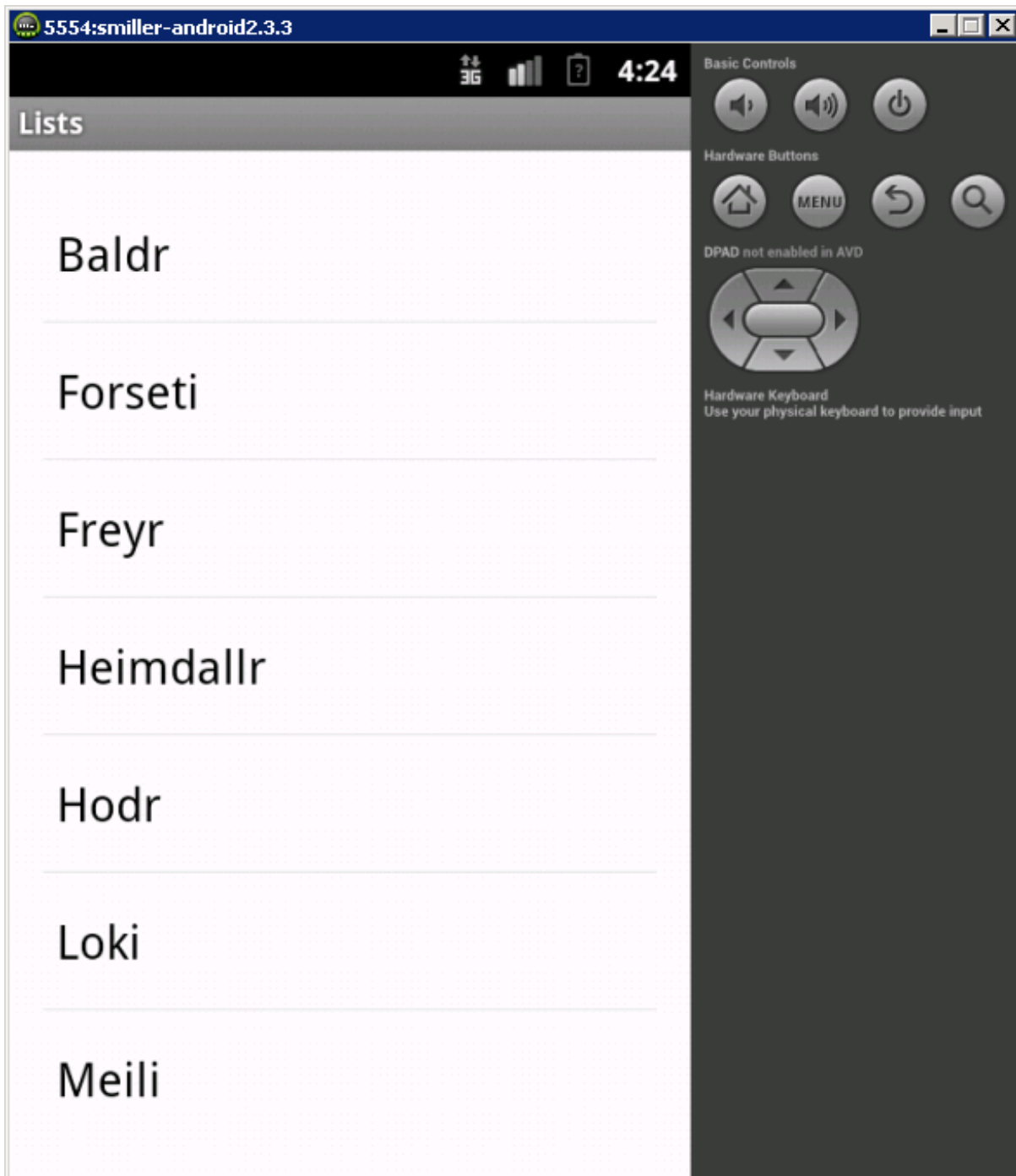
```
package com.oreillyschool.android1.lists;

import android.app.Activity;
import android.os.Bundle;
import android.widget.ArrayAdapter;

public class MainActivity extends ListActivity {
    private String[] data = new String [] {
        "Odin",
        "Thor",
        "Loki",
        ...
        "Forseti"
    };

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ArrayAdapter<String> adapter = new ArrayAdapter<String>(this, android.R.
layout.simple_list_item_1, android.R.id.text1, data);
        adapter.sort(new Comparator<String>() {
            @Override
            public int compare(String arg0, String arg1) {
                return arg0.compareTo(arg1);
            }
        });
        setListAdapter(adapter);
    }
}
```

Save and run it again. You'll see this:



Overriding ArrayAdapter

If all you need to display in your list is a simple view, the basic `ArrayAdapter` implementation is all you need. However, many Lists require a more involved layout to display a more complicated data model. There are some other default layouts that offer a bit more detail. For example, using `android.R.layout.simple_list_item_single_choice` includes a `CheckBox` in the list item layout. These default layouts can only get you so far though; in order to create a truly customized layout, you'll need to override the default implementation of your adapter. Let's do that now.

First, we'll need to create a new layout that will be used for each item in the List. Create a new Android XML layout named `my_list_item.xml`. Modify the XML as shown:

CODE TO TYPE:

```
<?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="match_parent"
    android:orientation="vertical" ->
    android:orientation="horizontal"
    android:gravity="center_vertical">

    <View
        android:id="@+id/color"
        android:layout_width="10dp"
        android:layout_height="50dp"
    />

    <TextView
        android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
    />

</LinearLayout>
```

Next, let's change our data model a little bit so that it contains more than just a String. We're going to create a class locally inside of MainActivity, but you could as easily create it in a new file:

CODE TO TYPE:

```
package com.oreillyschool.android1.lists;

import android.app.Activity;
import android.os.Bundle;
import android.widget.AdapterView;

public class MainActivity extends ListActivity {
    private String[] data = new String[] {
        "Odin",
        "Thor",
        "Loki",
        "Baldr",
        "Freyr",
        "Heimdallr",
        "Ullr",
        "Meili",
        "Hodr",
        "Forseti"
    };

    public class MyData {
        public String name;
        public boolean clicked;
        public MyData(String name) {
            this.name = name;
            this.clicked = false;
        }
    }

    private MyData[] data = new MyData[] {
        new MyData("Odin"),
        new MyData("Thor"),
        new MyData("Loki"),
        new MyData("Baldr"),
        new MyData("Freyr"),
        new MyData("Heimdallr"),
        new MyData("Ullr"),
        new MyData("Meili"),
        new MyData("Hodr"),
        new MyData("Forseti")
    };

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ArrayAdapter<String> adapter = new ArrayAdapter<String>(this, android.R.
layout.simple_list_item_1, android.R.id.text1, data);
        adapter.sort(new Comparator<String>() {
            @Override
            public int compare(String arg0, String arg1) {
                return arg0.compareTo(arg1);
            }
        });
        setListAdapter(adapter);
    }
}
```

You get an error; do you know why? Mull that over; we'll fix it in a minute. For now, let's move back to creating our customized Adapter. Create a new class in the **com.oreillyschool.android1.lists** package named **MyListAdapter**, and set the Superclass to **android.widget.AdapterView<MyData>**:

New Java Class

Java Class
Create a new Java class.

Source folder:

Package:

☐ Enclosing type:

Name:

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass:

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)
☐ Constructors from superclass
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))
☐ Generate comments

With the new class created and opened, make these changes to your code:

CODE TO TYPE:

```
package com.oreillyschool.android1.lists;

import android.widget.AdapterView;
import com.oreillyschool.android1.lists.MainActivity.MyData;
import android.content.Context;
import android.graphics.Color;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;

public class MyListAdapter extends ArrayAdapter<MyData> {

    private LayoutInflater inflater;

    public MyListAdapter(Context context, MyData[] data) {
        super(context, R.layout.my_list_item, data);
        inflater = LayoutInflater.from(context);
    }

    @Override
    public View getView(int position, View convertView, ViewGroup root) {
        View view = convertView;
        if (view == null) {
            view = inflater.inflate(R.layout.my_list_item, null);
        }
        MyData data = getItem(position);

        TextView textView = (TextView) view.findViewById(R.id.text);
        textView.setText(data.name);

        View imageView = view.findViewById(R.id.color);
        int color = data.clicked ? Color.RED : Color.BLUE;
        imageView.setBackgroundColor(color);

        return view;
    }
}
```

Now, we need one last change to hook up the new class to the ListView. Go back to MainActivity.java and make these changes:

CODE TO TYPE:

```
package com.oreillyschool.android1.lists;

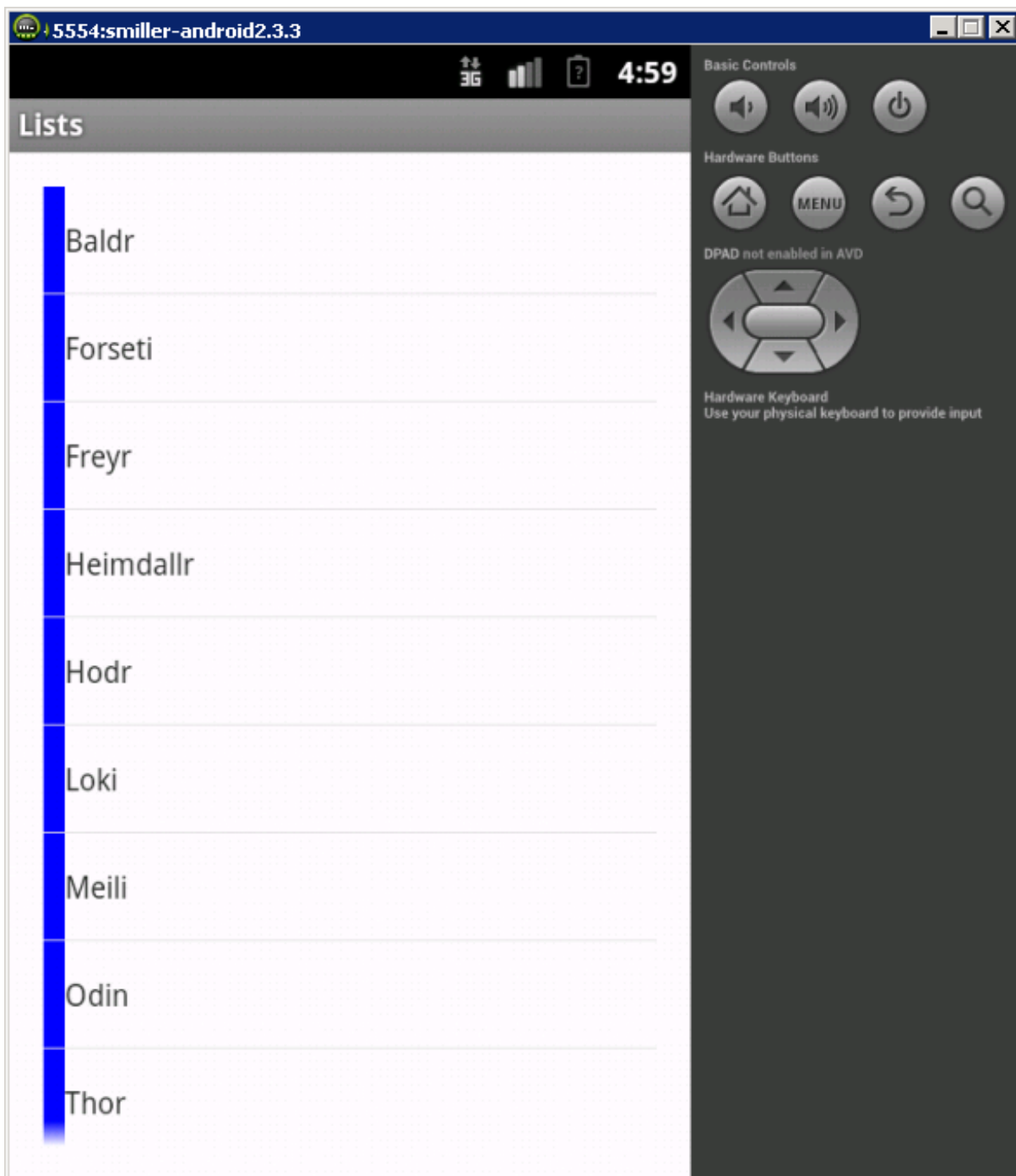
import android.app.Activity;
import android.os.Bundle;
import android.widget.AdapterView;

public class MainActivity extends ListActivity {
    public class MyData {
        public String name;
        public boolean clicked;
        public MyData(String name) {
            this.name = name;
            this.clicked = false;
        }
    }

    private MyData[] data = new MyData[] {
        new MyData("Odin"),
        new MyData("Thor"),
        new MyData("Loki"),
        new MyData("Baldr"),
        new MyData("Freyr"),
        new MyData("Heimdallr"),
        new MyData("Ullr"),
        new MyData("Meili"),
        new MyData("Hodr"),
        new MyData("Forseti")
    };

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ArrayAdapter<String> adapter = new ArrayAdapter<String>(this, android.R.
layout.simple_list_item_1, android.R.id.text1, data);
        adapter.sort(new Comparator<String>() {
            @Override
            public int compare(String arg0, String arg1) {
                return arg0.compareTo(arg1);
            }
        });
        MyListAdapter adapter = new MyListAdapter(this, data);
        adapter.sort(new Comparator<MyData>() {
            @Override
            public int compare(MyData arg0, MyData arg1) {
                return arg0.name.compareTo(arg1.name);
            }
        });
        setListAdapter(adapter);
    }
}
```

There's a lot going on here, but before we analyze our changes in detail, let's run it and make sure the code is working. Your list will look like this:



Now that we've verified that everything is working alright, let's take a closer look at our custom adapter, starting with the constructor:

OBSERVE:

```
public MyListAdapter(Context context, MyData[] data) {  
    super(context, R.layout.my_list_item, data);  
    inflater = LayoutInflater.from(context);  
}
```

In our constructor, we start off with the mandatory call to the **super constructor**; but we're calling a different super than we did before. And this time we're using a simpler constructor because we'll be handling the creation of each list item manually. The superclass will still handle the Array data, though:

OBSERVE:

```
@Override
public View getView(int position, View convertView, ViewGroup root) {
    View view = convertView;
    if (view == null) {
        view = inflater.inflate(R.layout.my_list_item, null);
    }
    MyData data = getItem(position);

    TextView textView = (TextView) view.findViewById(R.id.text);
    textView.setText(data.name);

    View imageView = view.findViewById(R.id.color);
    int color = data.clicked ? Color.RED : Color.BLUE;
    imageView.setBackgroundColor(color);

    return view;
}
```

The **getView()** method gets called each time the `ListView` needs to create a new `Listitem` or update an existing one. There are three parameters that get sent to the method, but our main concern is with the first two. The first, **int position**, is the position of the list item that needs a layout. The second, **View convertView**, can be either **null** or contain a recycled `View` component that was created previously with this method for a different item in the list. If the `convertView` is **null**, we'll need to inflate a new `View` component; otherwise we just need to update the data on the view component.

Inflating a layout can consume a fair amount of resources and time for the CPU. Inflating just one or two views isn't usually noticeable by a user, but a `ListView` can be scrolled very quickly, which means many different layouts could be required in very quick succession. For this reason, the Android `ListView` will recycle its items so that unnecessary layout inflation can be avoided.

Note

A common error programmers make when creating a custom list adapter is not updating all necessary values in a recycled view. Never rely on any default values of a newly inflated `View`, because if the view has been recycled, those values have very likely changed. If you find that your view is showing incorrect data on a list item, data that was perhaps on a previously rendered list item, then your first step to fixing that bug should be to verify you have invalidated all your custom data on your `View` in `getView`.

To get a reference to our data, we call the superclass method **getItem()**. This method returns an object of the type defined by the bound type parameter, which in our simple use-case is a `String`. This value type can be whatever data model you wish, just as long as your `Array` (or `ArrayList`) supplied to the super constructor contains objects only of that type.

The rest of the code probably looks familiar. It's the same type of code used to find views on an `Activity` and update the data for those views.

List Interaction

Now that we have the list data loading, let's update our code to react to a user clicking on a list item. Add the **MainActivity.java** method as shown:

MainActivity.java

```
package com.oreillyschool.android1.lists;

import android.app.Activity;
import android.os.Bundle;
import android.widget.ArrayAdapter;
import android.view.View;
import android.widget.ListView;

public class MainActivity extends ListActivity {
    public class MyData {
        public String name;
        public boolean clicked;
        public MyData(String name) {
            this.name = name;
            this.clicked = false;
        }
    }

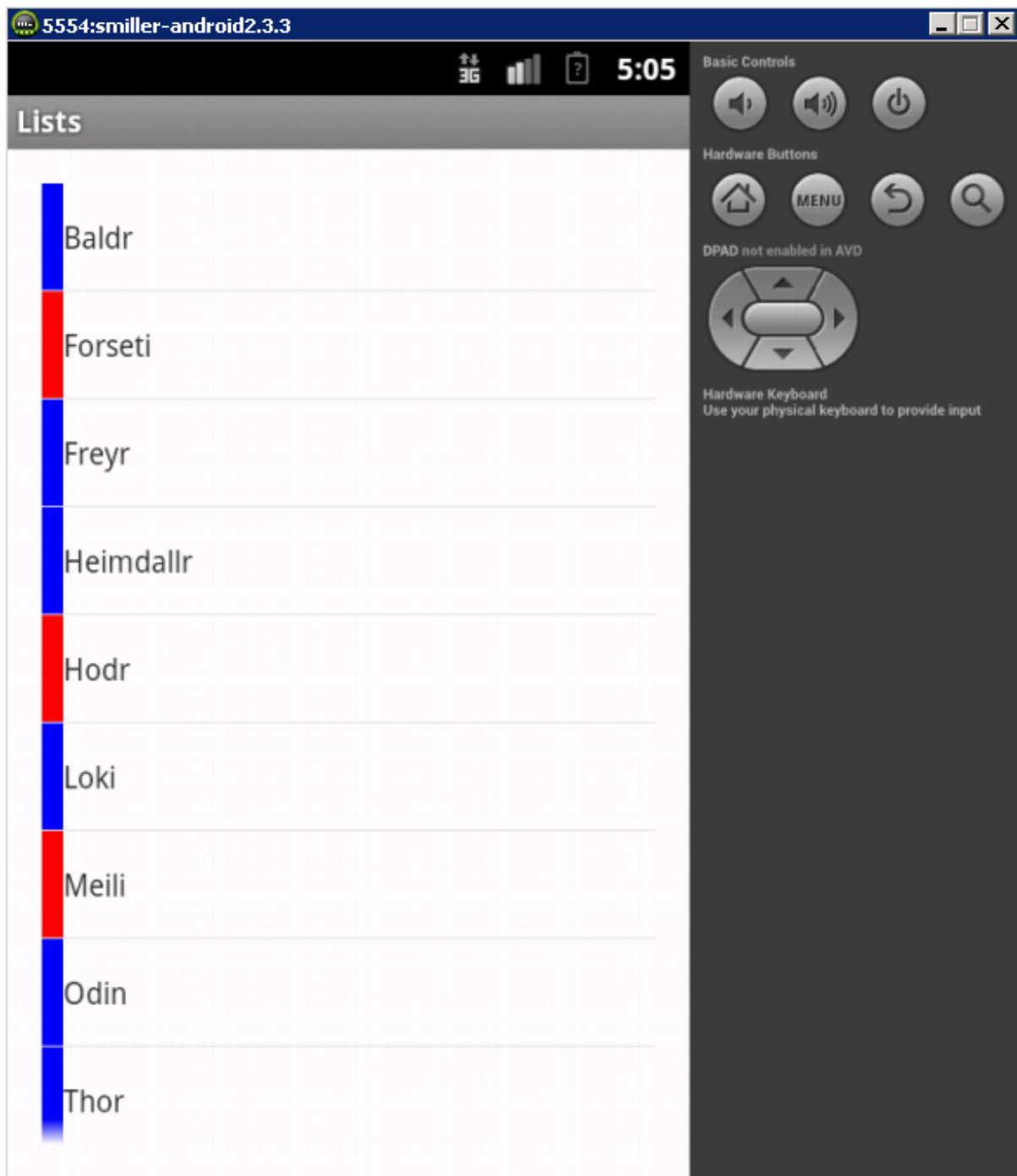
    private MyData[] data = new MyData[] {
        new MyData("Odin"),
        new MyData("Thor"),
        new MyData("Loki"),
        ...
        new MyData("Forseti")
    };

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        MyListAdapter adapter = new MyListAdapter(this, data);
        adapter.sort(new Comparator<MyData>() {
            @Override
            public int compare(MyData arg0, MyData arg1) {
                return arg0.name.compareTo(arg1.name);
            }
        });
        setListAdapter(adapter);
    }

    @Override
    protected void onItemClick(ListView l, View v, int position, long id) {
        MyListAdapter adapter = (MyListAdapter) getListAdapter();
        MyData item = adapter.getItem(position);
        item.clicked = !item.clicked;
        adapter.notifyDataSetChanged();
    }
}
```

This method override is available only to ListActivity. If you were handling your ListView component manually, you would need to set up a listener and send it to the **onItemClickListener()** method available on the ListView component.

Now, run the application one last time to test the results. Click some of the items to make sure the click handler is working:



Wrapping Up

We've covered a lot of ground here! It's good to know that you have a handle on ListView component and Adapters now, because virtually every Android application on the market uses at least one list. You're going to want to know how to use them well.

Alright then. So far, so good! See you next lesson...

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Dialogs, New and Old

Back for more, huh? Excellent. In this lesson, we'll talk about Dialogs. There are a few different types of dialogs, and also a few different methods to create and show Dialogs in Android.

There are essentially two ways to create dialogs in Android now, the new style and the old. With the release of Honeycomb, Android 3.x, which is specifically for tablets, Android added the new way of creating dialogs. This process has been carried over into Ice Cream Sandwich, Android 4.0, which works on both phones and tablets. Also, thanks to the Android support library, many features from Android 3+ are available to Android applications that target older builds of Android.

Old Style

Before we get going, let's create a new project for this lesson named **Dialogs**, assigned to the **Android1_Lessons** working set. Name the package **com.oreillyschool.android1.dialogs**.

The API methods used to create and show dialogs from an Activity are now marked "@Deprecated" in the most recent API updates to Android. However, it's still worth learning how to use these old methods in case you run across code that has already implemented dialogs that use it.

AlertDialog

There are a few different types of specialized Dialogs available in Android, such as AlertDialog and ProgressDialog. You can also create a custom Dialog class and populate it with whatever content you like. Let's begin by building an AlertDialog by adding a button to the primary view. Open **activity_main.xml** and make these changes:

```
/res/layout/activity_main.xml

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Old AlertDialog"
        android:onClick="onOldAlertDialogClick" />

</RelativeLayout>
```

Next, let's update the MainActivity to handle this button's click event; we'll tell the Activity we want to show a Dialog, and override the **onCreateDialog** method to create our Dialog. Make these changes:

MainActivity.java

```
package com.oreillyschool.android1.dialogs;

import android.app.Activity;
import android.app.AlertDialog;
import android.app.Dialog;
import android.os.Bundle;
import android.view.Menu;
import android.view.View;

public class MainActivity extends Activity {

    private static final int SIMPLE_DIALOG = 0;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void onOldAlertDialogClick(View view) {
        showDialog(SIMPLE_DIALOG);
    }

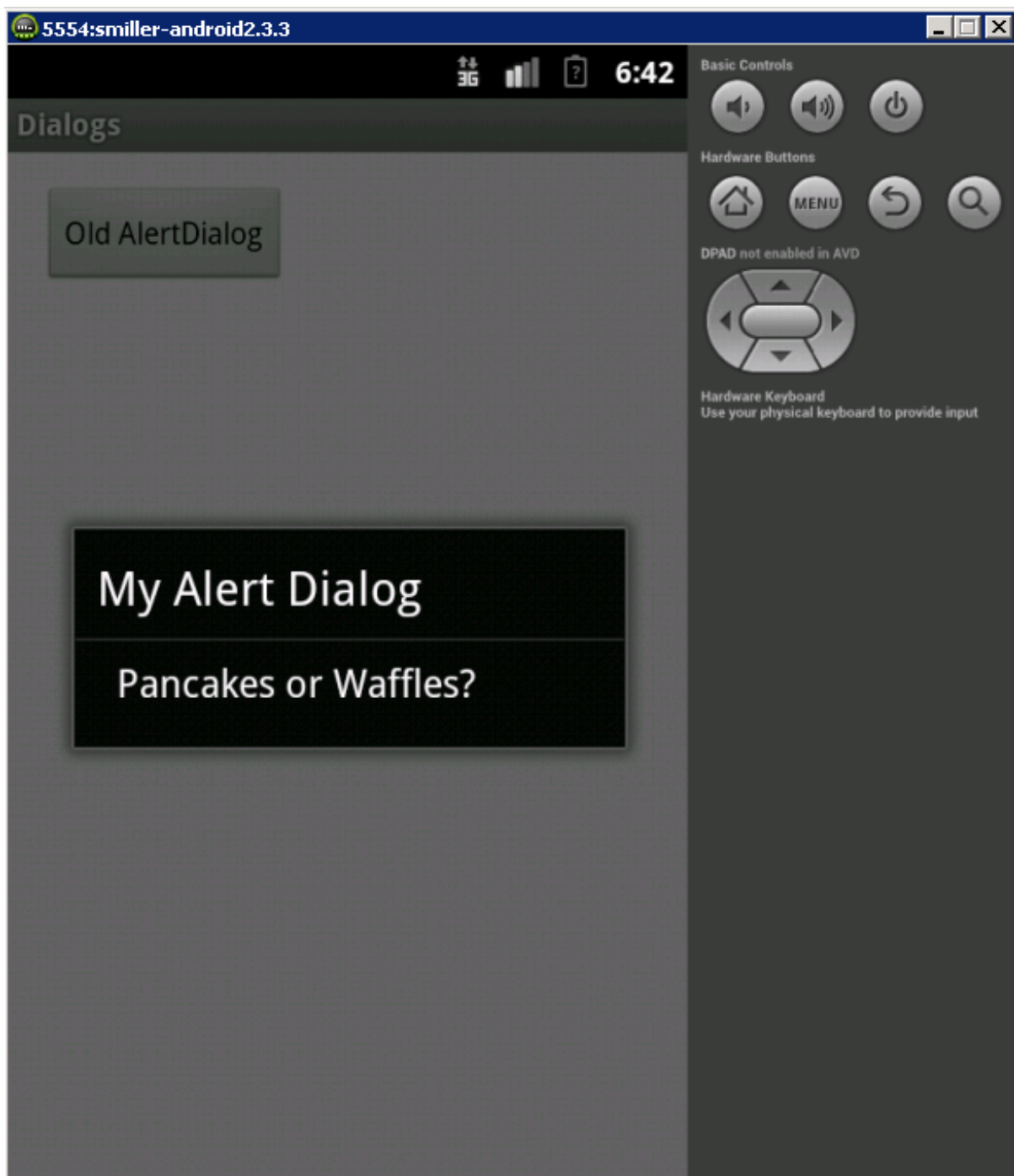
    @Override
    protected Dialog onCreateDialog(int id) {
        Dialog dialog = null;

        switch(id) {
            case SIMPLE_DIALOG:
                dialog = new AlertDialog.Builder(this).setTitle("My Alert Dialog")
                    .setMessage("Pancakes or Waffles?")
                    .create();
                break;
        }

        return dialog;
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }
}
```

Now save the changes and run the code to see what it does. Click the **OldAlertDialog** button and you'll see a dialog like this:



To close this dialog, you'll need to click the back button on the emulator. Typical dialogs contain at least one button to close, or multiple buttons to allow a user to make a decision. AlertDialog makes it easy to add one, two, or three buttons. Make these changes to **MainActivity.java**:

MainActivity.java

```
package com.oreillyschool.android1.dialogs;

import android.app.Activity;
import android.app.AlertDialog;
import android.app.Dialog;
import android.content.DialogInterface;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;

public class MainActivity extends Activity {

    private static final int SIMPLE_DIALOG = 0;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void onOldAlertDialogClick(View view) {
        showDialog(SIMPLE_DIALOG);
    }

    @Override
    protected Dialog onCreateDialog(int id) {
        Dialog dialog = null;

        switch(id) {
            case SIMPLE_DIALOG:
                dialog = new AlertDialog.Builder(this).setTitle("My Alert Dialog")
                    .setMessage("Pancakes or Waffles?")

                    .setNegativeButton("Boring Pancakes", dialogListener)
                    .setPositiveButton("Awesome Waffles!!", dialogListener)

                    .create();
                break;
        }

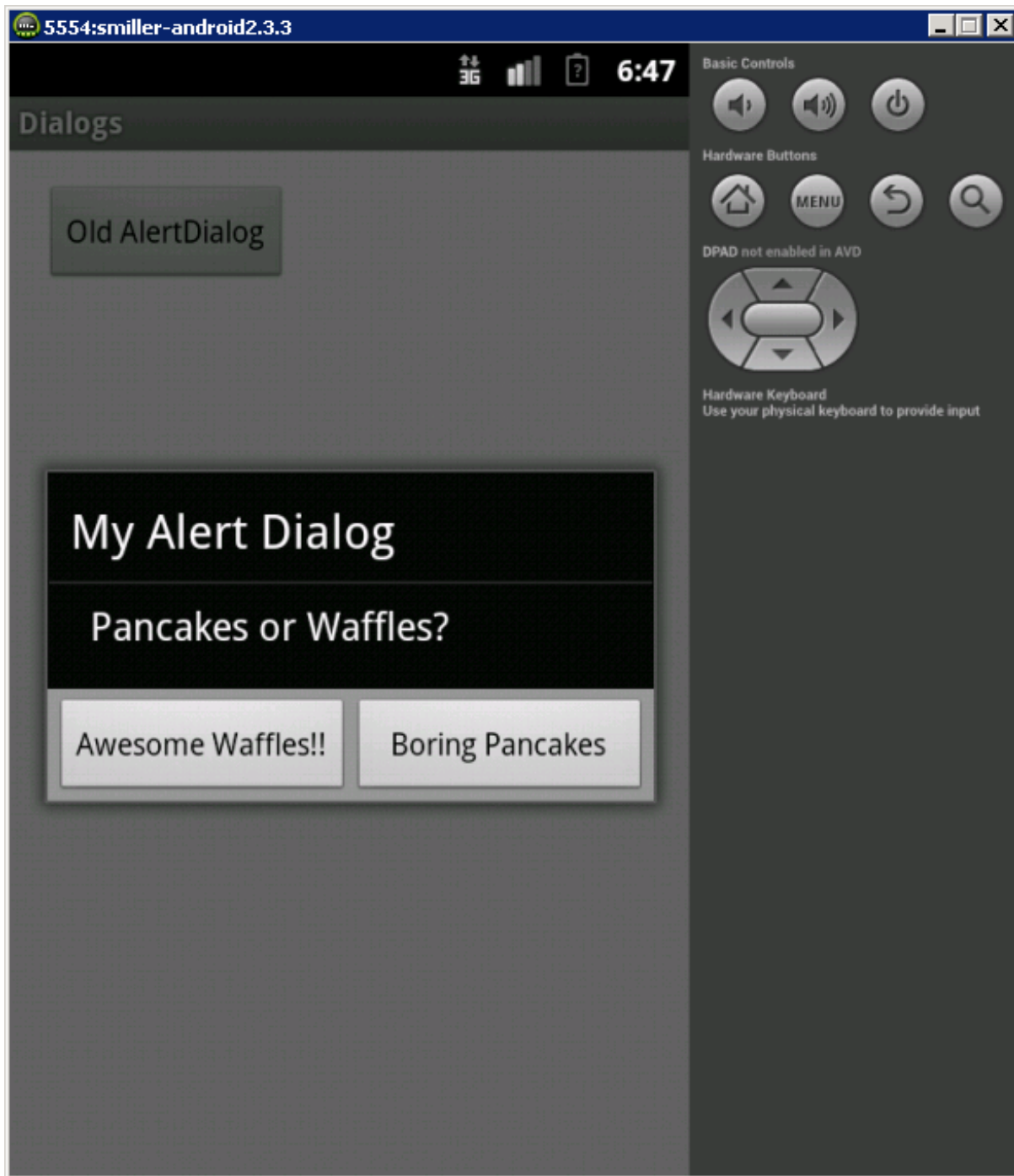
        return dialog;
    }

    private DialogInterface.OnClickListener dialogListener = new DialogInterface
    .OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int which) {
            switch (which) {
                case Dialog.BUTTON_NEGATIVE:
                    Toast.makeText(MainActivity.this, "Pancakes? Really?", Toast.LENGTH_LONG).show();
                    break;
                case Dialog.BUTTON_POSITIVE:
                    Toast.makeText(MainActivity.this, "Waffles are where it's at!", Toast.LENGTH_LONG).show();
                    break;
            }
            removeDialog(SIMPLE_DIALOG);
        }
    };
}
```

We defined the listener function we just used as a parameter.

Save and run this code to test the results. Now after clicking either button, you'll see two buttons in the dialog, as well as the appropriate Toast response. To close the dialog, we call the **removeDialog** method and send

the same id that was passed to **showDialog**. The dialog will close after either button click.



Note

In the Android SDK, there are two interface methods with the signature **OnClickListener()**. One is in the **android.view.View** package. It's the interface that we used earlier for handling clicks to buttons, and it's used for all standard view component click handling. The other is in the **android.content.DialogInterface** package, which is used for dialogs, and is the one we used in our code here. I like to define the type as **DialogInterface.OnClickListener** for dialog listeners to help differentiate between the more common **View.OnClickListener** methods; you can just as easily remove the **DialogInterface** portion from the type as long as you are sure to import the proper package.

The **onCreateDialog()** method that we implemented is called by an Activity only once per *id* parameter passed. If we want to make changes to the dialog created in **onCreateDialog** before the dialog is shown, we need to define a count variable and an **onPrepareDialog** method:

MainActivity.java

```
package com.oreillyschool.android1.dialogs;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.app.AlertDialog;
import android.app.Dialog;
import android.content.DialogInterface;
import android.widget.Toast;

public class MainActivity extends Activity {

    private static final int SIMPLE_DIALOG = 0;
    private int count = 0;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void onOldAlertDialogClick(View view) {
        showDialog(SIMPLE_DIALOG);
    }

    @Override
    protected Dialog onCreateDialog(int id) {
        Dialog dialog = null;

        switch(id) {
            case SIMPLE_DIALOG:
                dialog = new AlertDialog.Builder(this).setTitle("My Alert Dialog")
                    .setMessage("Pancakes or Waffles?")

                    .setNegativeButton("Boring Pancakes", dialogListener)
                    .setPositiveButton("Awesome Waffles!!", dialogListener)

                    .create();
                break;
        }

        return dialog;
    }

    @Override
    protected void onPrepareDialog(int id, Dialog dialog) {
        switch (id)
        {
            case SIMPLE_DIALOG:
                count++;
                dialog.setTitle("Dialog "+count);
                break;
        }
    }

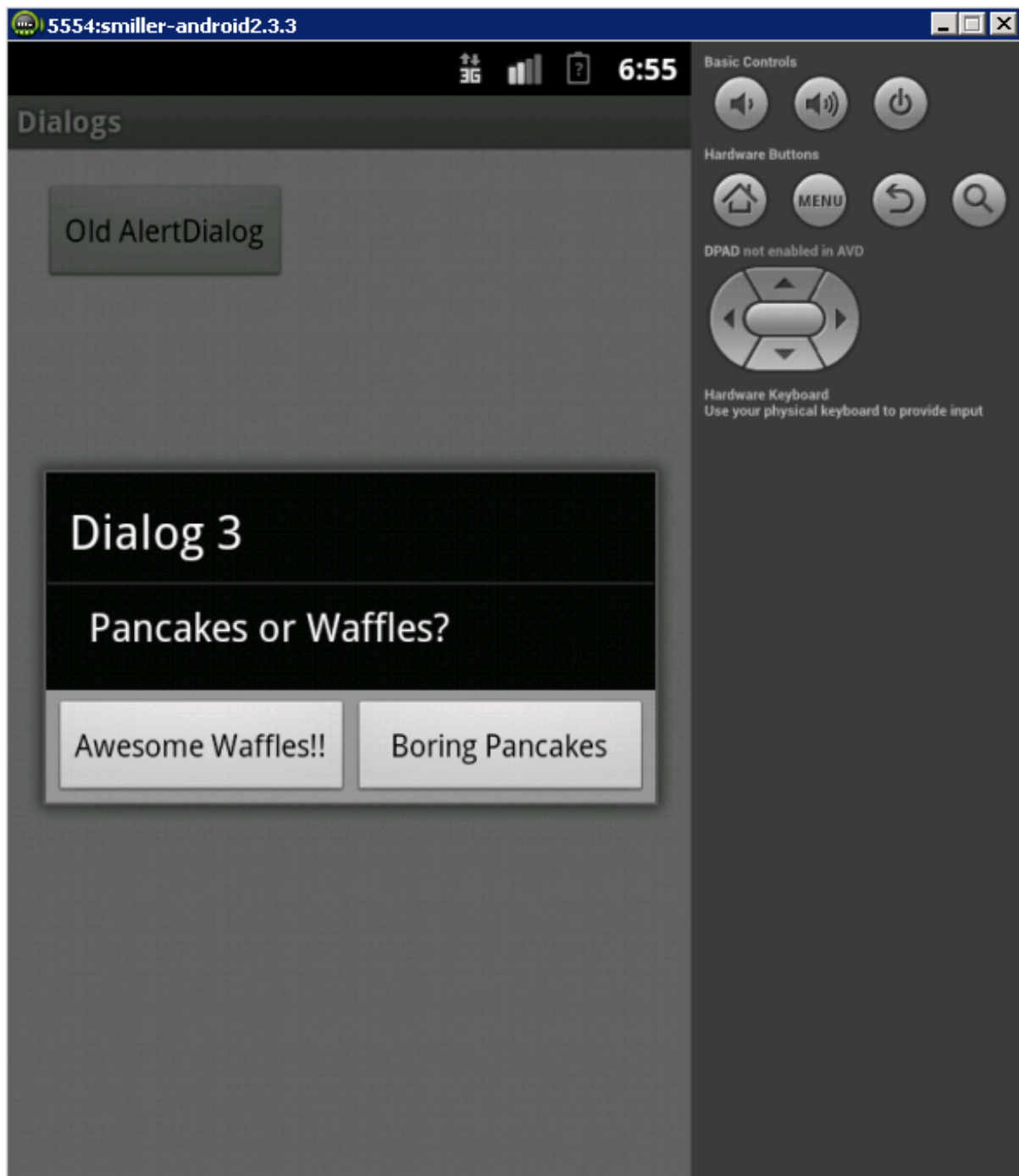
    private DialogInterface.OnClickListener dialogListener = new DialogInterface
    .OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int which) {
            switch (which) {
                case Dialog.BUTTON_NEGATIVE:
                    Toast.makeText(MainActivity.this, "Pancakes? Really?", Toast.LENGTH_LONG).show();
                    break;
                case Dialog.BUTTON_POSITIVE:
                    Toast.makeText(MainActivity.this, "Waffles are where it's at!",
```

```

Toast.LENGTH_LONG).show();
        break;
    }
    removeDialog(SIMPLE_DIALOG);
}
};
}

```

Save and run the program. The title of the dialog is replaced by the new title we defined in **onPrepareDialog** and the count is updated each time a new dialog is shown:



Custom Dialog

AlertDialog and its **Builder** class are handy tools for creating a quick and functional dialog, but if you need a customized view or more control over functionality, you'll need to create a custom dialog.

Let's create a new custom dialog that contains an image, a radio button, and a regular button. We'll start by creating a new button in the **activity_main.xml** view:

/res/layout/activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context=".MainActivity" >

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Old AlertDialog"
        android:onClick="onOldAlertDialogClick"
    />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Old CustomDialog"
        android:onClick="onOldCustomDialogClick"
    />

</LinearLayout>
```

Next, update **MainActivity.java** with the following changes:

MainActivity.java

```
package com.oreillyschool.android1.dialogs;

import android.app.Activity;
import android.app.AlertDialog;
import android.app.Dialog;
import android.content.DialogInterface;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;

public class MainActivity extends Activity {

    private static final int SIMPLE_DIALOG = 0;
    private static final int CUSTOM_DIALOG = 1;
    private int count = 0;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void onOldAlertDialogClick(View view) {
        showDialog(SIMPLE_DIALOG);
    }

    public void onOldCustomDialogClick(View view) {
        showDialog(CUSTOM_DIALOG);
    }

    @Override
    protected Dialog onCreateDialog(int id) {
        Dialog dialog = null;

        switch(id) {
            case SIMPLE_DIALOG:
                dialog = new AlertDialog.Builder(this).setTitle("My Alert Dialog")
                    .setMessage("Pancakes or Waffles?")

                    .setNegativeButton("Boring Pancakes", dialogListener)
                    .setPositiveButton("Awesome Waffles!!", dialogListener)

                    .create();
                break;
            case CUSTOM_DIALOG:
                dialog = new Dialog(this);
                dialog.setContentView(R.layout.custom_dialog);
                dialog.setTitle("My Custom Dialog");
                break;
        }

        return dialog;
    }

    @Override
    protected void onPrepareDialog(int id, Dialog dialog) {
        switch (id)
        {
            {
                case SIMPLE_DIALOG:
                    count++;
                    dialog.setTitle("Dialog "+count);
                    break;
            }
        }
    }

    private DialogInterface.OnClickListener dialogListener = new DialogInterface
```

```

.setOnClickListener() {
    @Override
    public void onClick(DialogInterface dialog, int which) {
        switch (which) {
            case Dialog.BUTTON_NEGATIVE:
                Toast.makeText(MainActivity.this, "Pancakes? Really?", Toast.LENGTH_LONG).show();
                break;
            case Dialog.BUTTON_POSITIVE:
                Toast.makeText(MainActivity.this, "Waffles are where it's at!", Toast.LENGTH_LONG).show();
                break;
        }
        removeDialog(SIMPLE_DIALOG);
    }
};
}

```

Finally, we'll need to create the layout for the dialog. Create a new XML Layout file in the **/res/layout** folder, name it **custom_dialog.xml**, and then make these changes to it:

/res/layout/custom_dialog.xml

```
<?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="match_parent"
    android:orientation="vertical" >

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center_vertical" >

        <ImageView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:src="@android:drawable/ic_menu_help" />

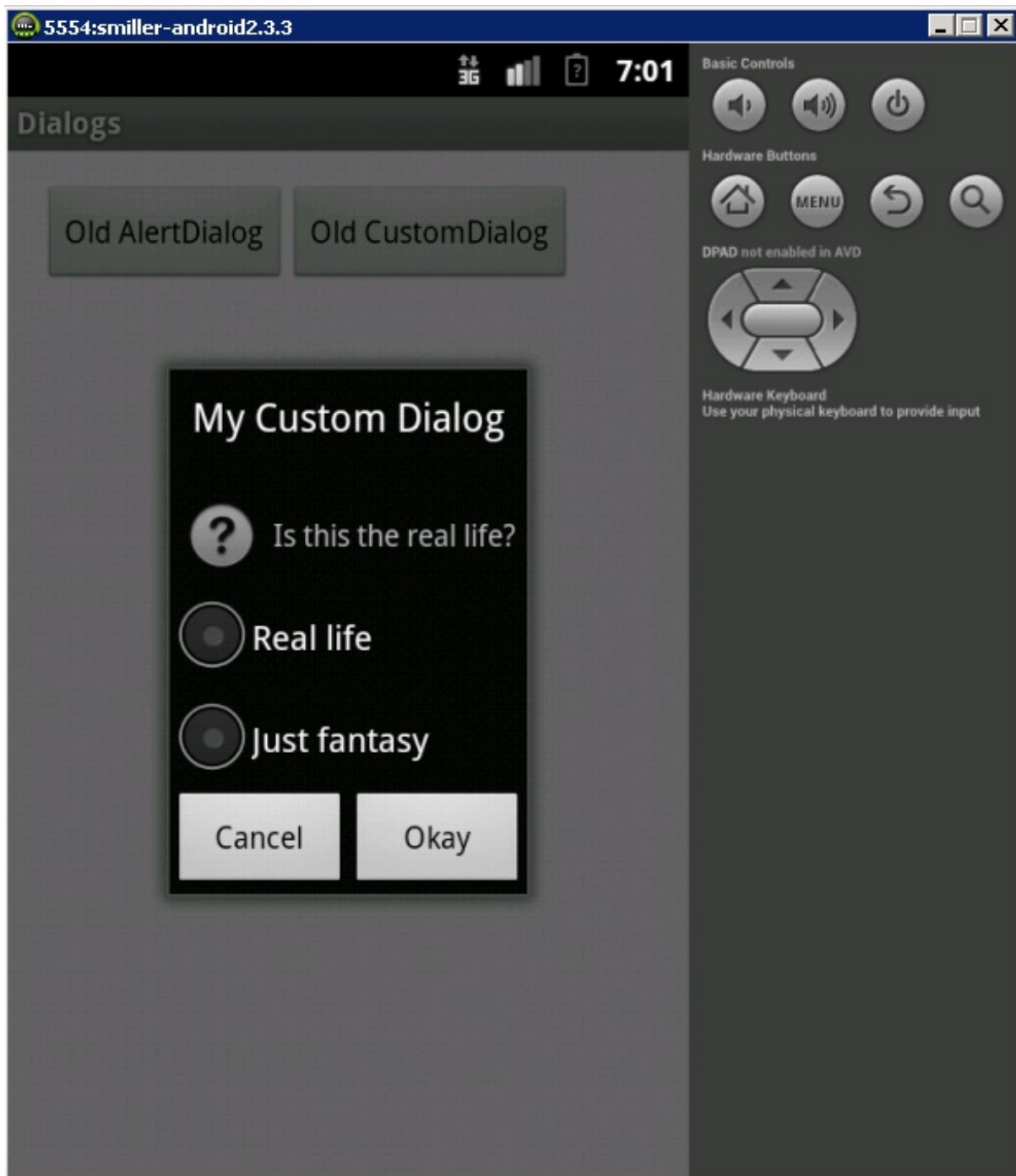
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Is this the real life?" />
    </LinearLayout>

    <RadioGroup
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical">
        <RadioButton
            android:id="@+id/rb_one"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Real life"/>
        <RadioButton
            android:id="@+id/rb_two"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Just fantasy"/>
    </RadioGroup>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <Button
            android:id="@+id/cancel_btn"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Cancel" />
        <Button
            android:id="@+id/okay_btn"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Okay" />
    </LinearLayout>

</LinearLayout>
```

Save all the changes and run the application to test the code. If all is working correctly, you'll see a new dialog like this:



Just like before, we haven't hooked up any event listeners to our dialog, so the only way to close it is by clicking the back button. Let's hook these buttons up to close the dialog. Make these changes to **MainActivity.java**:

MainActivity.java

```
package com.oreillyschool.android1.dialogs;

import android.app.Activity;
import android.app.AlertDialog;
import android.app.Dialog;
import android.content.DialogInterface;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.Toast;

public class MainActivity extends Activity {

    private static final int SIMPLE_DIALOG = 0;
    private static final int CUSTOM_DIALOG = 1;
    private int count = 0;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void onOldAlertDialogClick(View view) {
        showDialog(SIMPLE_DIALOG);
    }

    public void onOldCustomDialogClick(View view) {
        showDialog(CUSTOM_DIALOG);
    }

    @Override
    protected Dialog onCreateDialog(int id) {
        Dialog dialog = null;

        switch(id) {
            case SIMPLE_DIALOG:
                dialog = new AlertDialog.Builder(this).setTitle("My Alert Dialog")
                    .setMessage("Pancakes or Waffles?")

                    .setNegativeButton("Boring Pancakes", dialogListener)
                    .setPositiveButton("Awesome Waffles!!", dialogListener)

                    .create();
                break;
            case CUSTOM_DIALOG:
                dialog = new Dialog(this);
                dialog.setContentView(R.layout.custom_dialog);
                dialog.setTitle("My Custom Dialog");
                ((Button)dialog.findViewById(R.id.cancel_btn)).setOnClickListener(customDialogClickListener);
                ((Button)dialog.findViewById(R.id.okay_btn)).setOnClickListener(customDialogClickListener);
                break;
        }

        return dialog;
    }

    @Override
    protected void onPrepareDialog(int id, Dialog dialog) {
        switch (id)
        {
            case SIMPLE_DIALOG:
                count++;
                dialog.setTitle("Dialog "+count);
        }
    }
}
```



```

        break;
    }
}

private View.OnClickListener customDialogClickListener = new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        removeDialog(CUSTOM_DIALOG);
    }
};

private DialogInterface.OnClickListener dialogListener = new DialogInterface.OnClickListener() {
    @Override
    public void onClick(DialogInterface dialog, int which) {
        switch (which) {
            case Dialog.BUTTON_NEGATIVE:
                Toast.makeText(MainActivity.this, "Pancakes? Really?", Toast.LENGTH_LONG).show();
                break;
            case Dialog.BUTTON_POSITIVE:
                Toast.makeText(MainActivity.this, "Waffles are where it's at!", Toast.LENGTH_LONG).show();
                break;
        }
        removeDialog(SIMPLE_DIALOG);
    }
};
}

```

Test the code to make sure it works. Both buttons should close the dialog. You might've noticed that this time we used the **View.OnClickListener** instead of the **DialogInterface.OnClickListener** interface. That's because we're setting the listener directly on the button in the view of our Dialog, and not on the Dialog itself. In fact, the base Dialog class doesn't even use the **DialogInterface.OnClickListener** interface, it uses only the subclasses like **AlertDialog** and **ProgressDialog**.

Note

You might be tempted to try and use the **android:onClick** attribute shortcut in the XML to handle the click event from the buttons in the dialog. Unfortunately, this won't work inside a dialog and will actually throw an error when you click the button. That's because the view is a part of the Dialog and not the Activity, and the Dialog class does not implement a dynamic **android:onClick** listener.

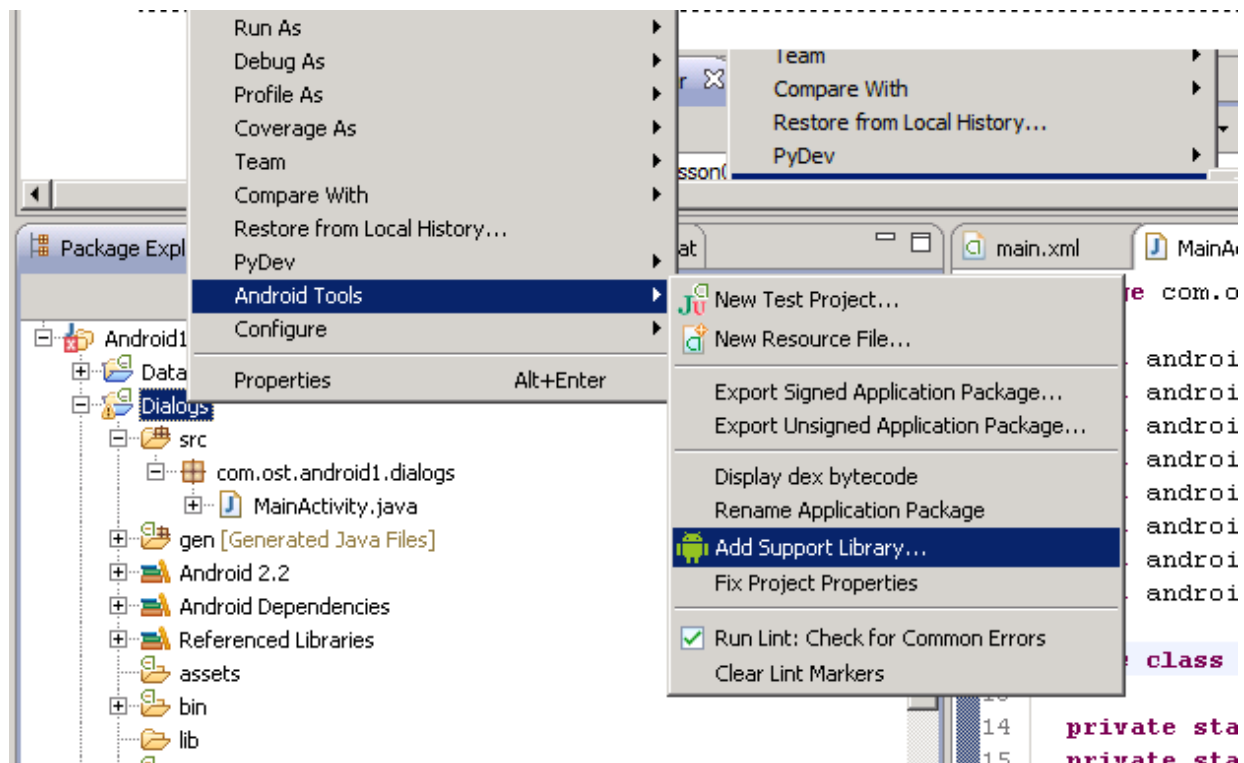
New Style

As of Android 3.0 and beyond, there is a new standard way to create and manage Android dialogs. But there is a way to implement Dialogs using the new processes and still target an older version of the Android SDK. Before we can get to that though, we're going to have to take a short detour and talk about the Android compatibility library and fragments.

Support Library

Unfortunately, the majority of Android phones in use run older versions of Android and, as such, they don't have access to the latest Android SDK features. In order to help developers take advantage of the new features of the latest SDK, Google created a "compatibility library" (also called "support library" or "support package") that allows applications that target older versions of the Android SDK to take advantage of many of the new features of Android.

To use the Support library we will need to download and add it to our project. ADT makes this process very easy now. Right-click on the root project folder **Dialogs**, choose **Android Tools | Add Support Library**, select the latest support library version when prompted, click **Accept**, and you're done. Eclipse will automatically download the latest version of the support library and include it in your project. When it's finished you can verify the process worked by expanding the *libs* folder to find the *android-support-v4.jar* file.



Fragments

Perhaps the most important feature in the Support Library is its support for fragments. A Fragment is like a mini-activity. They behave much like Activities, but they must be added to an activity. They can be created and destroyed within an Activity's lifecycle, but if the Activity is stopped, no fragment within it can be started; if the Activity is destroyed, then all of its fragments are destroyed as well.

Let's get a little practice in with basic fragments. We'll convert our current dialog application to use a fragment. Start by creating a new class named **MainFragment** and have it extend **Fragment**:

New Java Class

Java Class
Create a new Java class.

Source folder: Dialogs/src Browse...

Package: com.ost.android.dialogs Browse...

☐ Enclosing type: Browse...

Name: MainFragment

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass: android.support.v4.app.Fragment Browse...

Interfaces: Add...
Remove

Which method stubs would you like to create?

☐ public static void main(String[] args)
☐ Constructors from superclass
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))
☐ Generate comments

? Finish Cancel

Now let's create a new XML Layout File named **main_fragment.xml**, copy all the XML from **activity_main.xml**, and place it in **main_fragment.xml**. It will look like this when you are finished:

/res/layout/main_fragment.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context=".MainActivity" >

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Old AlertDialog"
        android:onClick="onOldAlertDialogClick" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Old CustomDialog"
        android:onClick="onOldCustomDialogClick" />

</LinearLayout>
```

Now close **main_fragment.xml**, go back to **activity_main.xml**, and update it to use the fragment **MainFragment**. We'll also change it to a basic **FrameLayout** since we'll only be using a single fragment:

/res/layout/activity_main.xml

```
<LinearLayoutFrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity" >>

<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Old AlertDialog"
    android:onClick="onOldAlertDialogClick" />

<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Old CustomDialog"
    android:onClick="onOldCustomDialogClick" />

    <fragment
        class="com.ost.android1.dialogs.MainFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</LinearLayoutFrameLayout>
```

Save that file. Now, back in **MainFragment.java**, update it to load a view:

MainFragemnt.java

```
package com.oreillyschool.android1.dialogs;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class MainFragemnt extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        return inflater.inflate(R.layout.activity_main_fragment, container, false);
    }
}
```

This is a little different from how we define a view in an Activity, but it's actually similar to how we create views in a list adapter. In a fragment that has a view, you need to override and implement the **onCreateView** method. An **Inflater** object is passed to the method as a convenience to inflate a new View. The **Inflater.inflate** method is actually overloaded, but the version we're using here takes three parameters: a resource id to the view layout, a **ViewGroup** object that will be used to parent the View, and a **Boolean** to determine whether to attach the View to the ViewGroup. One of the overloaded methods doesn't have the 3rd **Boolean** parameter, effectively making that an optional parameter. We don't want to attach our View to the container during inflation, so we send "false," that way the inflated view inherits only the layout parameters from the container.

Before we can test this code we'll need to make one last change to MainActivity so it can load fragments (it's only a small change so we'll just show the affected lines this time):

MainActivity.java

```
import android.app.Activity;
import android.app.AlertDialog;
import android.app.Dialog;
import android.content.DialogInterface;
import android.os.Bundle;
import android.support.v4.app.FragmentActivity;
import android.view.View;
import android.widget.Button;
import android.widget.Toast;

public class MainActivity extends FragmentActivity {
    ...
}
```

Save and run the Application. Everything functions exactly as it did before. We'll explore the benefits and best usage for fragments in future lessons, but right now we'll move on, or rather get back, to creating dialogs in the "new way" with a type of fragment called **DialogFragment**.

DialogFragment

A **DialogFragment** is a type of fragment that's loaded into an Activity like any other fragment, except that its view is a dialog. This allows the dialog's lifecycle to be tied to a fragment instance instead of directly to the Activity itself. This also lets us create dialogs that support the standard for Android devices running SDK 3.0 and up.

Let's add and implement a DialogFragment to our application. Create a new class called **MyCustomDialogFragment** and have it extend the **DialogFragment** class:

New Java Class

Java Class
Create a new Java class.

Source folder: Dialogs/src Browse...

Package: com.ost.android.dialogs Browse...

☐ Enclosing type: Browse...

Name: MyCustomDialogFragment

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass: android.support.v4.app.DialogFragment Browse...

Interfaces: Add...
Remove

Which method stubs would you like to create?

☐ public static void main(String[] args)
☐ Constructors from superclass
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))
☐ Generate comments

? Finish Cancel

Next, let's update this code to use the same view as our previous custom dialog:

MyCustomDialogFragment.java

```
package com.oreillyschool.android1.dialogs;

import android.app.Dialog;
import android.os.Bundle;
import android.support.v4.app.DialogFragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.Button;

public class MyCustomDialogFragment extends DialogFragment {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setStyle(DialogFragment.STYLE_NORMAL, android.R.style.Theme_Dialog);
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View v = inflater.inflate(R.layout.custom_dialog, container, false);
        ((Button)v.findViewById(R.id.cancel_btn)).setOnClickListener(listener);
        ((Button)v.findViewById(R.id.okay_btn)).setOnClickListener(listener);
        return v;
    }

    private View.OnClickListener listener = new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            dismiss();
        }
    };

    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        Dialog d = super.onCreateDialog(savedInstanceState);
        d.setTitle("My New Custom Dialog");
        return d;
    }
}
```

Now, open **main_fragment.xml** and add another button:

main_fragment.xml

```
...

<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Old CustomDialog"
    android:onClick="onOldCustomDialogClick"
/>

<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="New CustomDialog"
    android:onClick="onNewCustomDialogClick"
/>

</LinearLayout>
```

And finally, go back to **MainActivity.java** to implement the click event for this button:

MainActivity.java

```
import android.app.AlertDialog;
import android.app.Dialog;
import android.content.DialogInterface;
import android.os.Bundle;
import android.support.v4.app.DialogFragment;
import android.support.v4.app.FragmentActivity;
import android.support.v4.app.FragmentManager;
import android.support.v4.app.FragmentTransaction;
import android.view.View;
import android.widget.Button;
import android.widget.Toast;

public class MainActivity extends FragmentActivity {
    private static final int SIMPLE_DIALOG = 0;
    private static final int CUSTOM_DIALOG = 1;

    private static final String CUSTOM_DIALOG_FRAGMENT = "customDialogFragment";

    private int count = 0;

    ...

    public void onOldCustomDialogClick(View view) {
        showDialog(CUSTOM_DIALOG);
    }

    public void onNewCustomDialogClick(View view) {
        FragmentManager fm = getSupportFragmentManager();
        FragmentTransaction ft = fm.beginTransaction();
        DialogFragment df = new MyCustomDialogFragment();
        df.show(ft, CUSTOM_DIALOG_FRAGMENT);
    }

    ...
}
```

Save all this code and run it. The new button launches a custom dialog that looks exactly the same as the previous custom dialog (since they're using the same layout view), only with a slightly different title.

Now instead of using the Activity to manage the display and concealment of the Dialog, we use the DialogFragment through the **DialogFragment.show()** and **DialogFragment.dismiss()** methods. **DialogFragment.show()** requires either a **FragmentManager** object or a **FragmentTransaction**. An instance of **FragmentManager** can always be retrieved from a **FragmentActivity** by calling the **getFragmentManager()** method in Android SDK version 3.0 and up, or by calling **getSupportFragmentManager()** which is available to the **FragmentActivity** in the support library for use in applications using older versions of Android SDK.

We could've created an AlertDialog using a DialogFragment as well. The only difference between this and the old method is that we wouldn't even need to implement **onCreateView()** for the AlertDialog. All we would have to do is copy the **AlertDialog.Builder** code from before into the **onCreateDialog** implementation of the dialog fragment.

Wrapping Up

Wow! We covered a lot in this lesson too. We learned about Dialogs and dipped our toes into the support library and fragments. Don't be discouraged if you still find fragments perplexing. The more we use them, the more you'll understand them.

See you in the next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Menus

Hi there and welcome back! In this lesson we'll cover the various types of menus in Android. Menus are convenient tools that provide contextual options to a user during an activity. There's quite a bit to cover so let's get started!

Menus, Menus, Menus

The term "menu" can refer to a few different types of components in Android. The most common menu in Android is the *Options Menu*. The Options Menu appears when a user touches the hardware menu button on their device. In versions before Android 3.0, the Options Menu appears as a small window anchored to the bottom of the device screen, holding up to six menu items (automatically arranged into two rows of three buttons). On devices using Android 3.0 and up, the menu is integrated into the Application Bar.

Another type of menu is the *Context Menu*. Unlike the Options Menu, the Context Menu is directly associated with a View component instead of the Activity. A Context Menu will appear when the user long-presses on the View with which it was registered. The Context Menu is similar in appearance to a traditional Dialog. These are typically implemented on items in a list to allow the user to perform an alternate action on the list item.

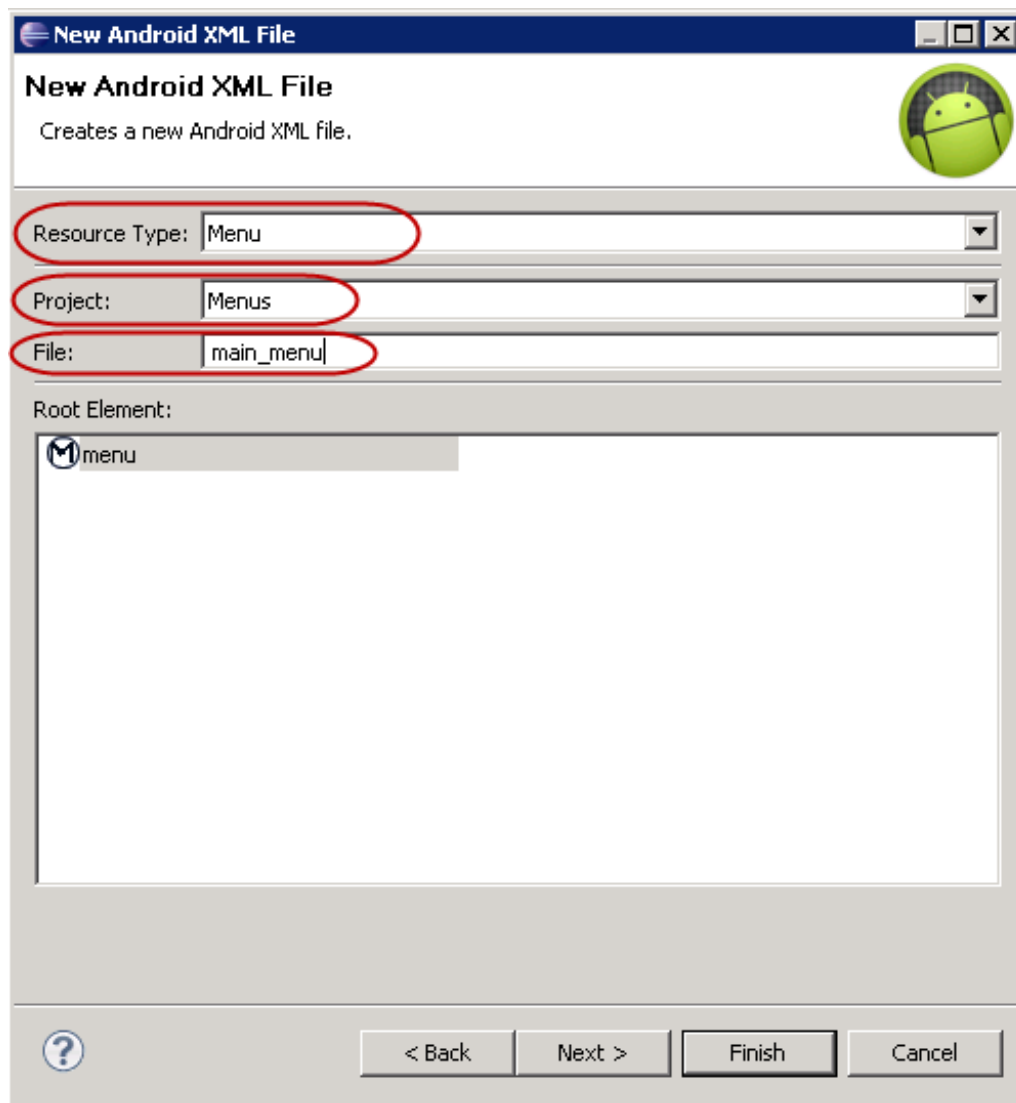
The last type of menu is a *Submenu*. A Submenu is a menu item contained within another menu. A Submenu can be added to an Options Menu or a Context Menu. Regardless of the type of menu to which a Submenu is attached, it will resemble a Context Menu.

Options Menu

To get started making our menus, create a new project named **Menus**, with the with the Package name **com.oreillyschool.android1.menus**, in the **Android1_Lessons** working set.

Menus in Android are typically defined using XML resource files that are stored in the **/res/menu** folder of the project. Let's use the ADT XML values file wizard to create our menu:

1. Select the Menus project and then select **File | New | Other** (or use the keyboard shortcut **Ctrl-N**).
2. In the "Select a Wizard" dialog, choose the **Android XML File** option in the **Android** folder, and click **Next**.
3. In the "New Android XML File" wizard, change the "Resource Type" to **Menu**, choose the **Menus** project, enter the file name **main_menu**. Click **Finish** to create the XML resource.



If it did not already exist, the wizard creates the **/res/menu** folder automatically and saves the new XML resource into that folder. Now let's get into that new XML and create some menu items! Modify **main_menu.xml** as shown:

```
/res/menu/main_menu.xml

<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android" >

    <item
        android:id="@+id/mi_dalek"
        android:title="Dalek"/>

    <item
        android:id="@+id/mi_cybermen"
        android:title="Cybermen"/>

</menu>
```

In order to define menus in the XML resource, the root xml node must be **menu**, and its children must be either **item** or **group** nodes. An item node typically takes no children, but has many possible attributes, for instance, **id**, **title**, **icon**, or **visible**. Each item node represents an item in the menu. A group node can take only other item nodes as children; a group node is used to define certain attributes for its child item nodes, such as **visible**, **enabled**, and **checkable**.

Our menu here has only two items defined with titles (and no icons).

Note

There's a plethora of native icons available to developers in the Android SDK. There are many common menu actions among apps on Android (like "info" and "help"). If you implement these kinds of menu items in your applications, I recommend that you use system icons. System icons provide a consistent experience for the end user that will help them to understand how your application works. If you find yourself in need of custom icons, check out the [Android recommended guidelines](#) for icon design.

We've hard-coded our strings for the title attribute, but we could have used a `res/string.xml` string reference id instead (using the `@string/<string-id>` format). In fact, just about any Android function that takes a string can also take a reference to string resource id. I generally recommend using string resources rather than hard-coded strings. So far in the course, we've been hard-coding most of our strings to keep the code concise and focused. But in your own future projects, you'll want to put all strings that will be visible to users in the strings XML resource file.

Okay, now let's get started with **MainActivity.java** to implement the menu resource. Add the code below to **MainActivity.java** as shown:

MainActivity.java

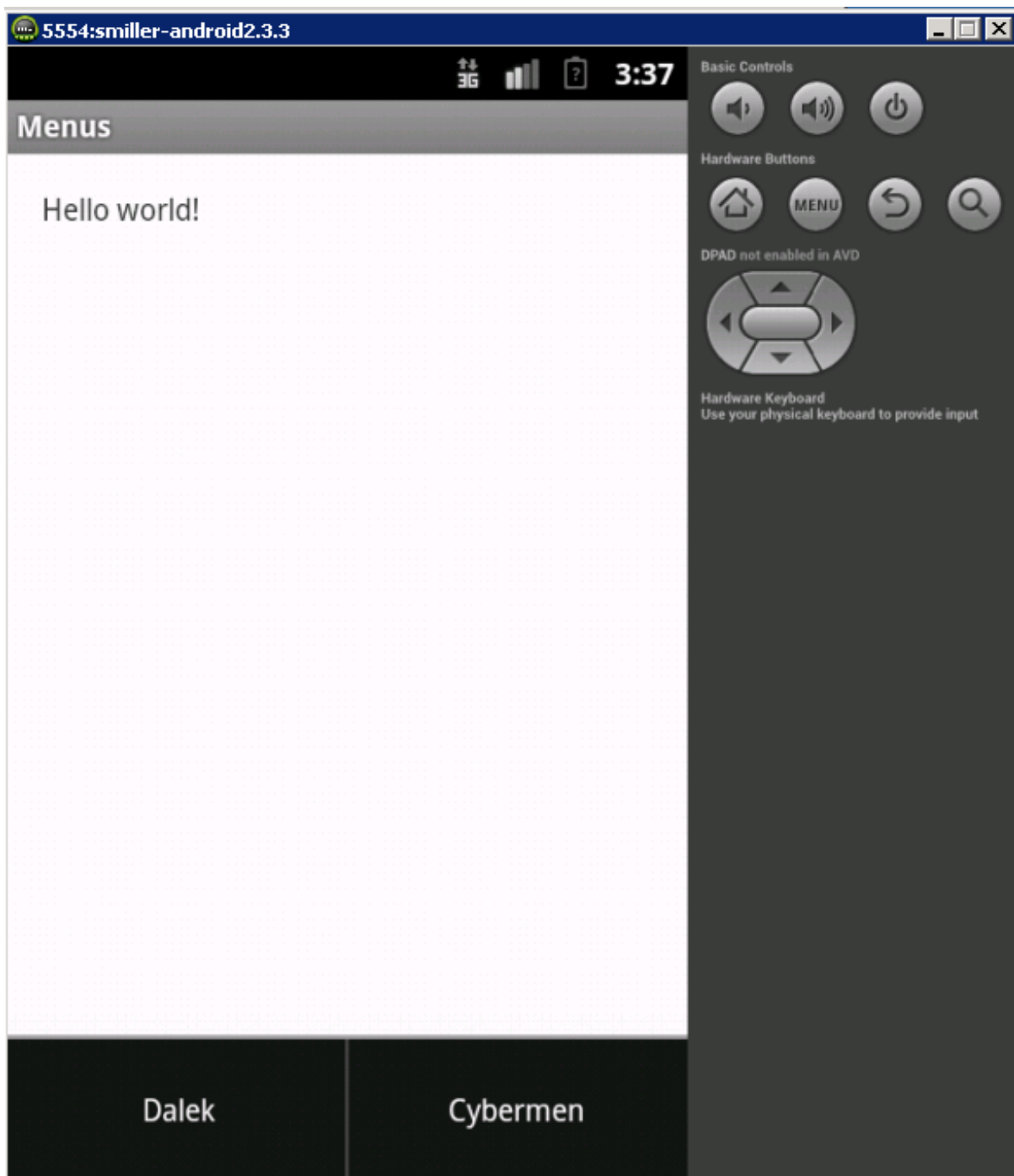
```
package com.oreillyschool.android1.menus;

import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuInflater;

public class MainActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
        final MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.main_menu, menu);
        return true;
    }
}
```

And that's it! Now we can save and run the project. Once the application is installed and running on the emulator, click the **Menu** key on the emulator screen; the menu should pop up from the bottom:



OBSERVE:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    final MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.main_menu, menu);
    return true;
}
```

We use the `onCreateOptionsMenu()` method to create our menu. The method is called by Android automatically when a user clicks on the Menu button. `onCreateOptionsMenu()` receives one object in its parameters, a `Menu` object. Then we inflate the menu object with our XML by using a `MenuInflater` obtained from `getMenuInflater()`. The menu inflater takes two parameters: an `R.java` reference to the XML file and the `menu object` into which the XML is inflated.

Just like the dialog methods for creating a dialog, `onCreateOptionsMenu()` is called only once by the Activity during its active lifecycle. To make updates to the menu before it is presented to the user, we would override and implement the `onPrepareOptionsMenu()` method. We'll practice doing that later in the lesson,

but for now let's add some code to **MainActivity.java** in order to respond to clicks on the menu items:

MainActivity.java

```
package com.oreillyschool.android1.menus;

import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.widget.Toast;

public class MainActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        final MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.main_menu, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case R.id.mi_dalek:
                Toast.makeText(this, "Exterminate!", Toast.LENGTH_LONG).show();
                break;
            case R.id.mi_cybermen:
                Toast.makeText(this, "You will be upgraded.", Toast.LENGTH_SHORT
).show();
                break;
        }
        return true;
    }
}
```

Save and run it. You'll see the appropriate toast message pop up for each menu item.

OBSERVE:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.mi_dalek:
            Toast.makeText(this, "Exterminate!", Toast.LENGTH_LONG).show();
            break;
        case R.id.mi_cybermen:
            Toast.makeText(this, "You will be upgraded.", Toast.LENGTH_SHORT
).show();
            break;
    }
    return true;
}
```

When a menu item is selected, the Android system triggers a call to the **onOptionsItemSelected()** method, passing the selected **MenuItem** object as the only parameter. As long as you have assigned an **id** attribute to each of the items in your XML, you can use a switch/cases block on the **item.getItemId()** integer to find out which item was clicked and respond accordingly. The **third parameter** for the **makeText()** method controls how long the message will display.

To demonstrate the "More" button, we'll add a few more items to **main_menu.xml**:

/res/menu/main_menu.xml

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

    <item
        android:id="@+id/mi_dalek"
        android:title="Dalek"/>

    <item
        android:id="@+id/mi_cybermen"
        android:title="Cybermen"/>

    <item
        android:id="@+id/mi_angels"
        android:title="Weeping Angels"/>

    <item
        android:id="@+id/mi_silence"
        android:title="The Silence"/>

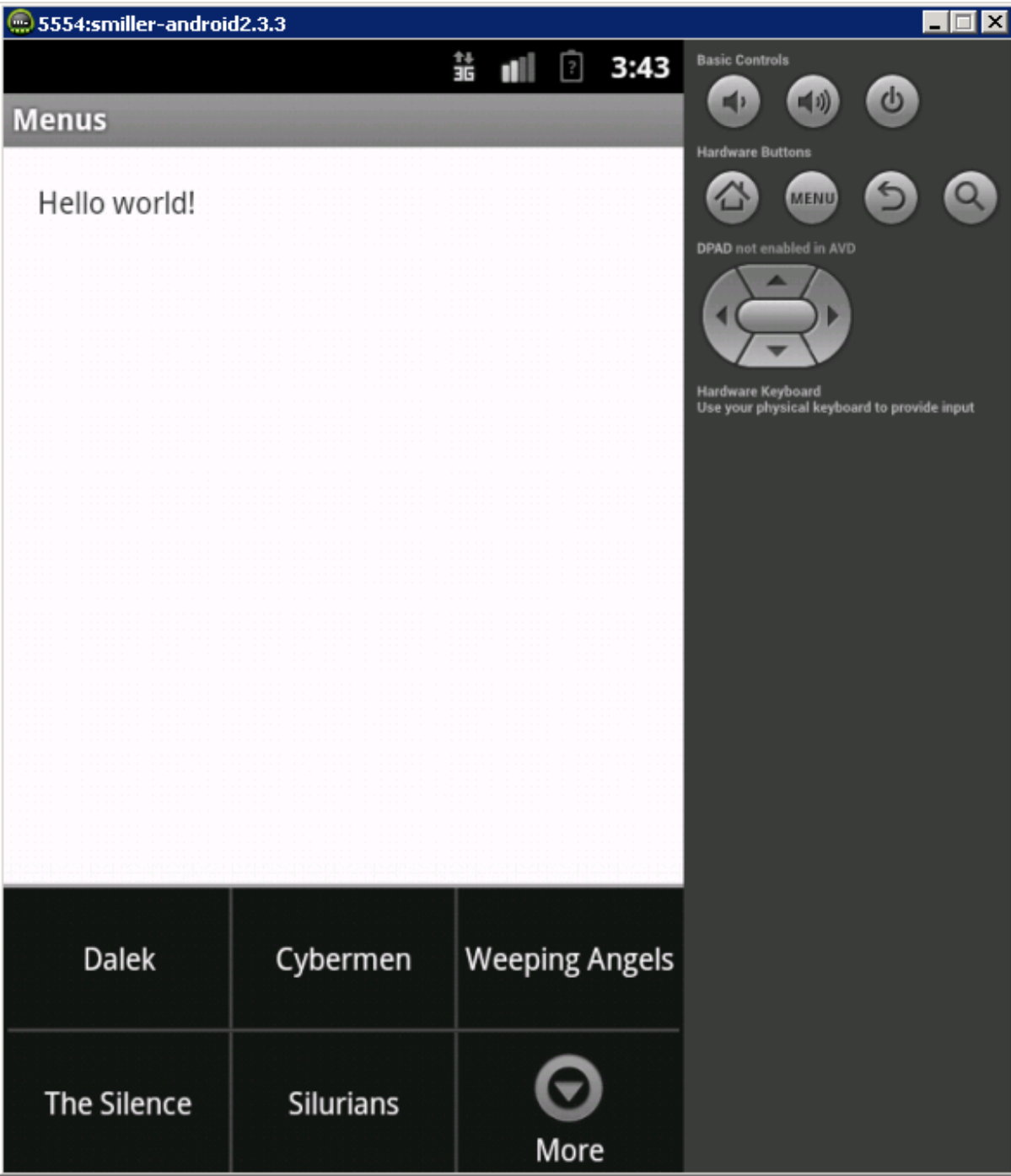
    <item
        android:id="@+id/mi_silurians"
        android:title="Silurians"/>

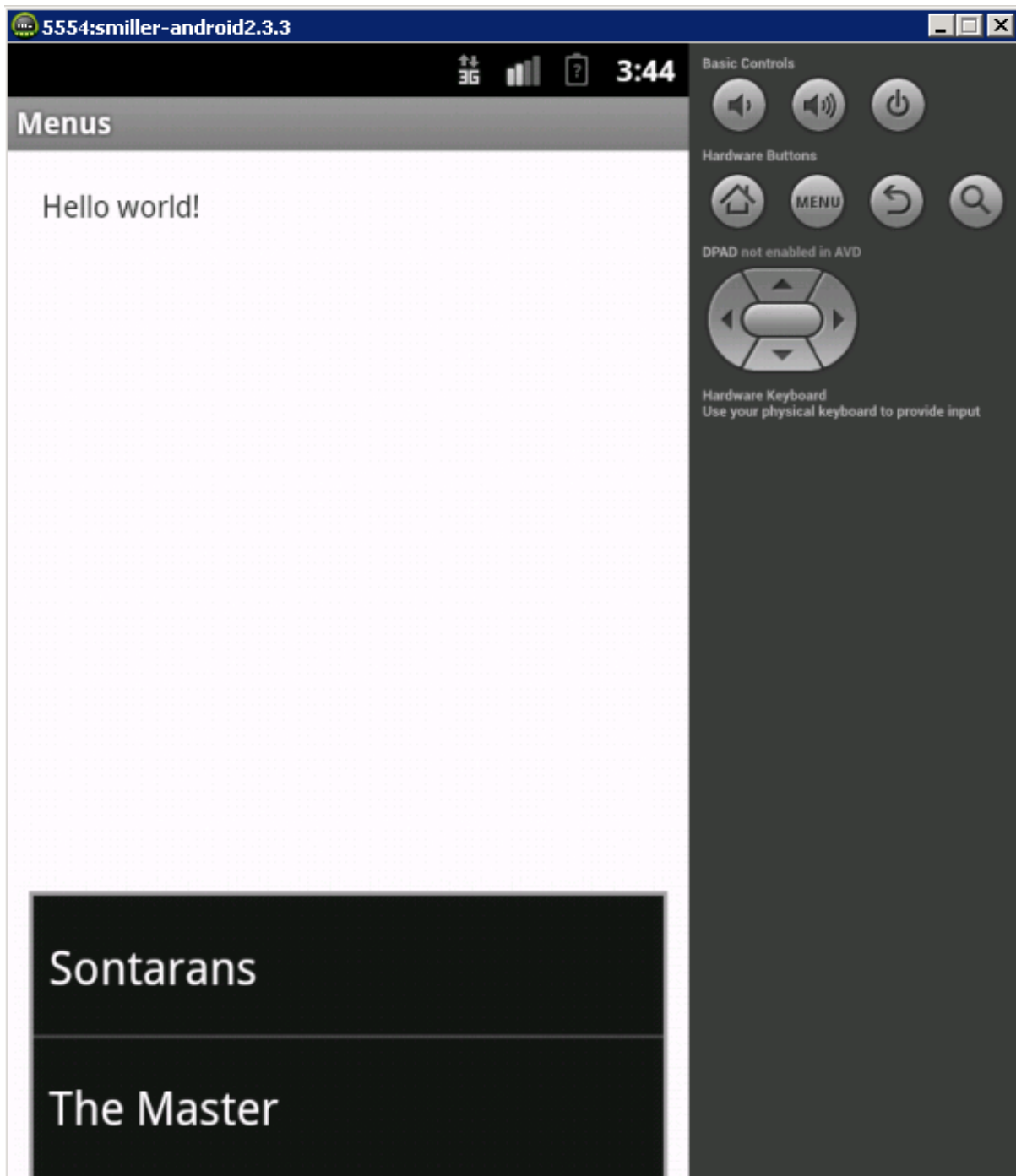
    <item
        android:id="@+id/mi_sontarans"
        android:title="Sontarans"/>

    <item
        android:id="@+id/mi_master"
        android:title="The Master"/>

</menu>
```

Now when you run the application you'll see the regular Options Menu showing only the first five items; the sixth item is a "More" button. When you click the More button, you'll see another vertically arranged menu that contains the remaining items:





We can try experimenting with a submenu now as well. Modify **main_menu.xml** as shown (in Eclipse, you can indent a block of code by highlighting it and pressing the **Tab** key):

/res/menu/main_menu.xml

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

    <item
        android:id="@+id/mi_dalek"
        android:title="Dalek"/>

    <item
        android:id="@+id/mi_cybermen"
        android:title="Cybermen"/>

    <item
        android:title="Others">

        <menu>

            <item
                android:id="@+id/mi_angels"
                android:title="Weeping Angels"/>

            <item
                android:id="@+id/mi_silence"
                android:title="The Silence"/>

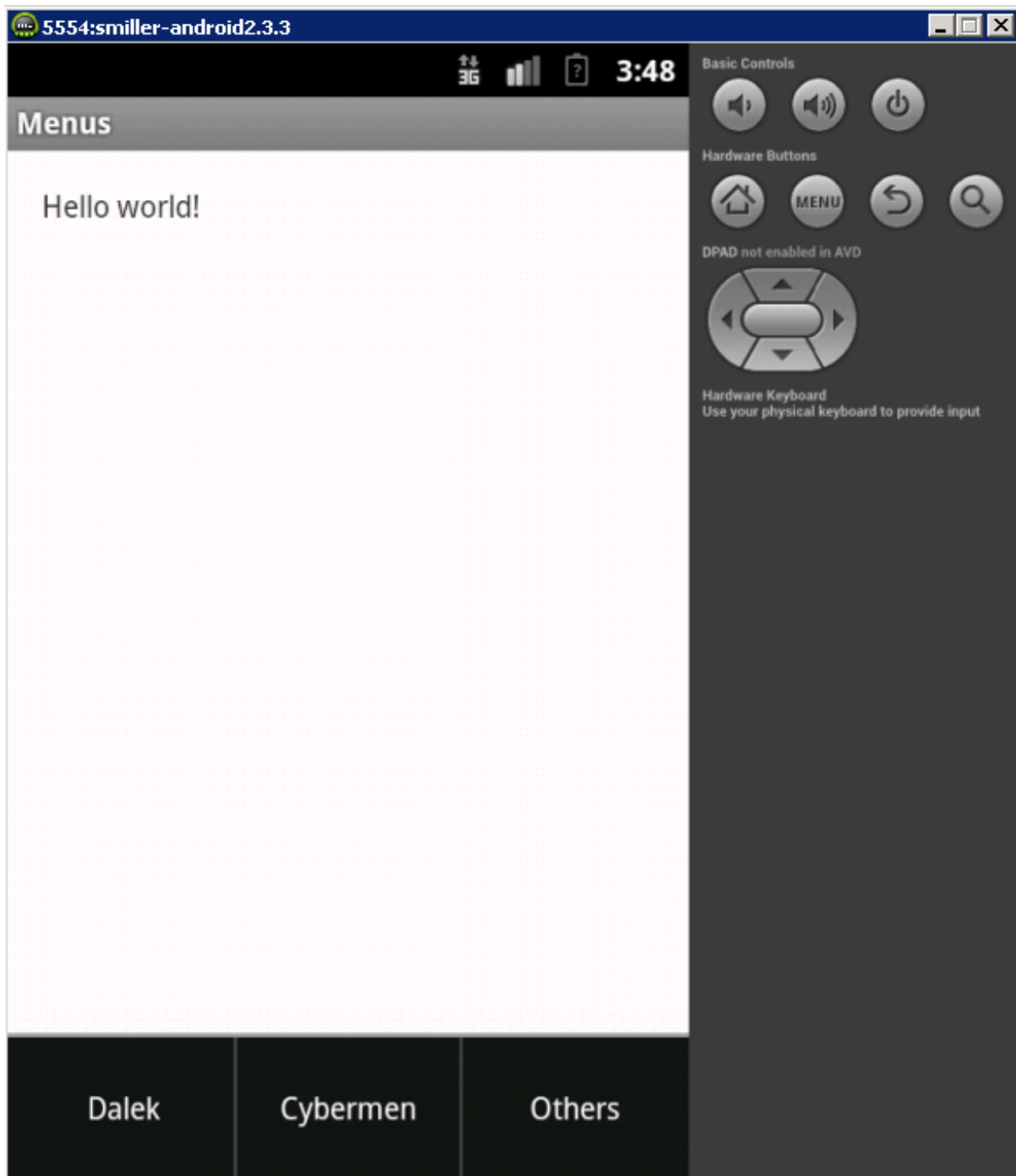
            <item
                android:id="@+id/mi_silurians"
                android:title="Silurians"/>

            <item
                android:id="@+id/mi_sontarans"
                android:title="Sontarans"/>

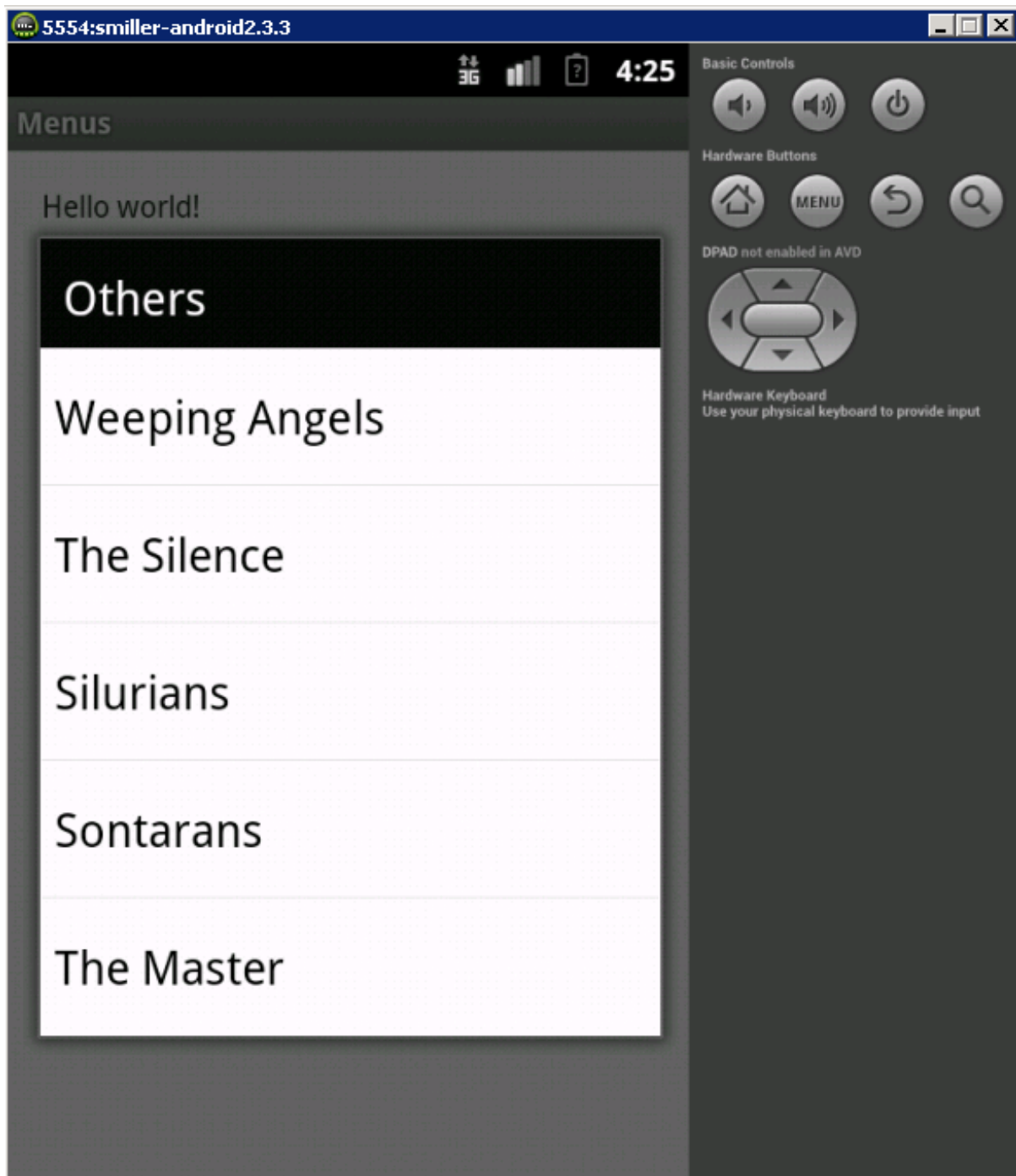
            <item
                android:id="@+id/mi_master"
                android:title="The Master"/>

        </menu>
    </item>
</menu>
```

Here we wrapped some of the previous items in a **menu** tag, and then wrapped that within a new **item** tag titled "Others." Now when we test the application menu button, the initial Options Menu only shows "Dalek," "Cybermen," and "Others."



When you click **Others**, you see a submenu with the remaining items:



Modifying an Options Menu

As you do with a Dialog, you'll want to update a menu before it becomes visible to the user. Since the "create" method for a menu gets called just once during the lifecycle of an activity, you need to use another method to handle the updates. Make these changes to **MainActivity.java**:

MainActivity.java

```
package com.oreillyschool.android1.menus;

import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.widget.Toast;

public class MainActivity extends Activity {
    private int optionLastClickedId = -1;
    private int optionClickedId = -1;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        final MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.main_menu, menu);
        return true;
    }

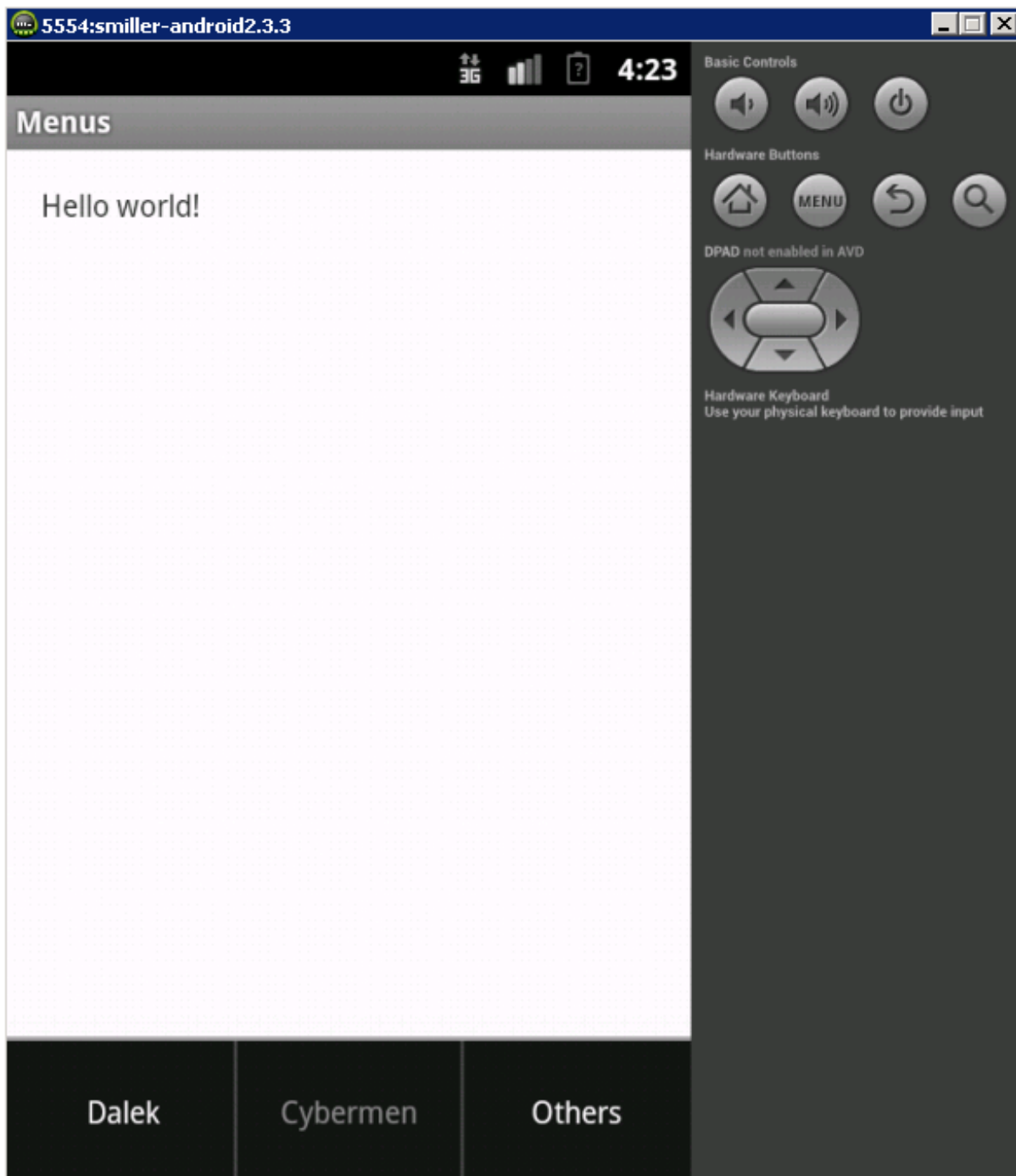
    @Override
    public boolean onPrepareOptionsMenu(Menu menu) {
        MenuItem item = menu.findItem(optionLastClickedId);
        if (item != null) {
            item.setEnabled(true);
        }
        item = menu.findItem(optionClickedId);
        if (item != null) {
            item.setEnabled(false);
        }
        optionLastClickedId = optionClickedId;

        return super.onPrepareOptionsMenu(menu);
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        optionClickedId = item.getItemId();

        switch (optionClickedId) {
switch (item.getItemId()) {
            case R.id.mi_dalek:
                Toast.makeText(this, "Exterminate!", Toast.LENGTH_LONG).show();
                break;
            case R.id.mi_cybermen:
                Toast.makeText(this, "You will be upgraded.", Toast.LENGTH_SHORT).sh
ow();
                break;
        }
        return true;
    }
}
```

Save and run it. Whichever menu item you click will become disabled the next time the Options Menu is presented (and the previously disabled item will become enabled):



OBSERVE:

```
public boolean onPrepareOptionsMenu(Menu menu) {  
    MenuItem item = menu.findItem(optionLastClickedId);  
    if (item != null) {  
        item.setEnabled(true);  
    }  
    item = menu.findItem(optionClickedId);  
    if (item != null) {  
        item.setEnabled(false);  
    }  
    optionLastClickedId = optionClickedId;  
  
    return super.onPrepareOptionsMenu(menu);  
}
```

The `onPrepareOptionsMenu()` method receives one parameter, the `Menu` object that was created earlier in the `onCreateOptionsMenu()` method. We can use this object to modify the menu any way we like, such as

finding **specific MenuItem objects** and **modifying their properties**, or even adding or removing a MenuItem from the Menu.

Note

Menus can be created programmatically as well, using the **Menu** and **MenuItem** constructors, then adding them with any of the various "add" methods available on **Menu**. In this lesson, though, we create all of our Menus with the most commonly used **MenuInflater** method.

Context Menu

Like Options Menus, Context Menus can also be defined using an XML resource file. We can reuse the menu XML resource that we used earlier for the Options Menu to implement a Context Menu. In order to do that, we register the menu with a view component in our Activity. First, we'll need to define a view component in **activity_main.xml**:

/res/layout/activity_main.xml

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context=".MainActivity" >

    <TextView
        android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello" />

</RelativeLayout>
```

We use the default TextView that was generated with our project, and we add an **id** attribute so we can locate the component. Next let's update **MainActivity.java** to register the view with a menu:

MainActivity.java

```
package com.oreillyschool.android1.menus;

import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.widget.Toast;
import android.view.ContextMenu;
import android.view.ContextMenu.ContextMenuInfo;
import android.view.View;

public class MainActivity extends Activity {
    private int optionLastClickedId = -1;
    private int optionClickedId = -1;

    /** Called when the activity is first created. */
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        View textView = findViewById(R.id.text);
        registerForContextMenu(textView);
    }

    @Override
    public void onCreateContextMenu(ContextMenu menu, View v, ContextMenuInfo menuInfo) {
        super.onCreateContextMenu(menu, v, menuInfo);
        final MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.main_menu, menu);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        final MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.main_menu, menu);
        return true;
    }

    @Override
    public boolean onPrepareOptionsMenu(Menu menu) {
        MenuItem item = menu.findItem(optionLastClickedId);
        if (item != null) {
            item.setEnabled(true);
        }
        item = menu.findItem(optionClickedId);
        if (item != null) {
            item.setEnabled(false);
        }
        optionLastClickedId = optionClickedId;

        return super.onPrepareOptionsMenu(menu);
    }

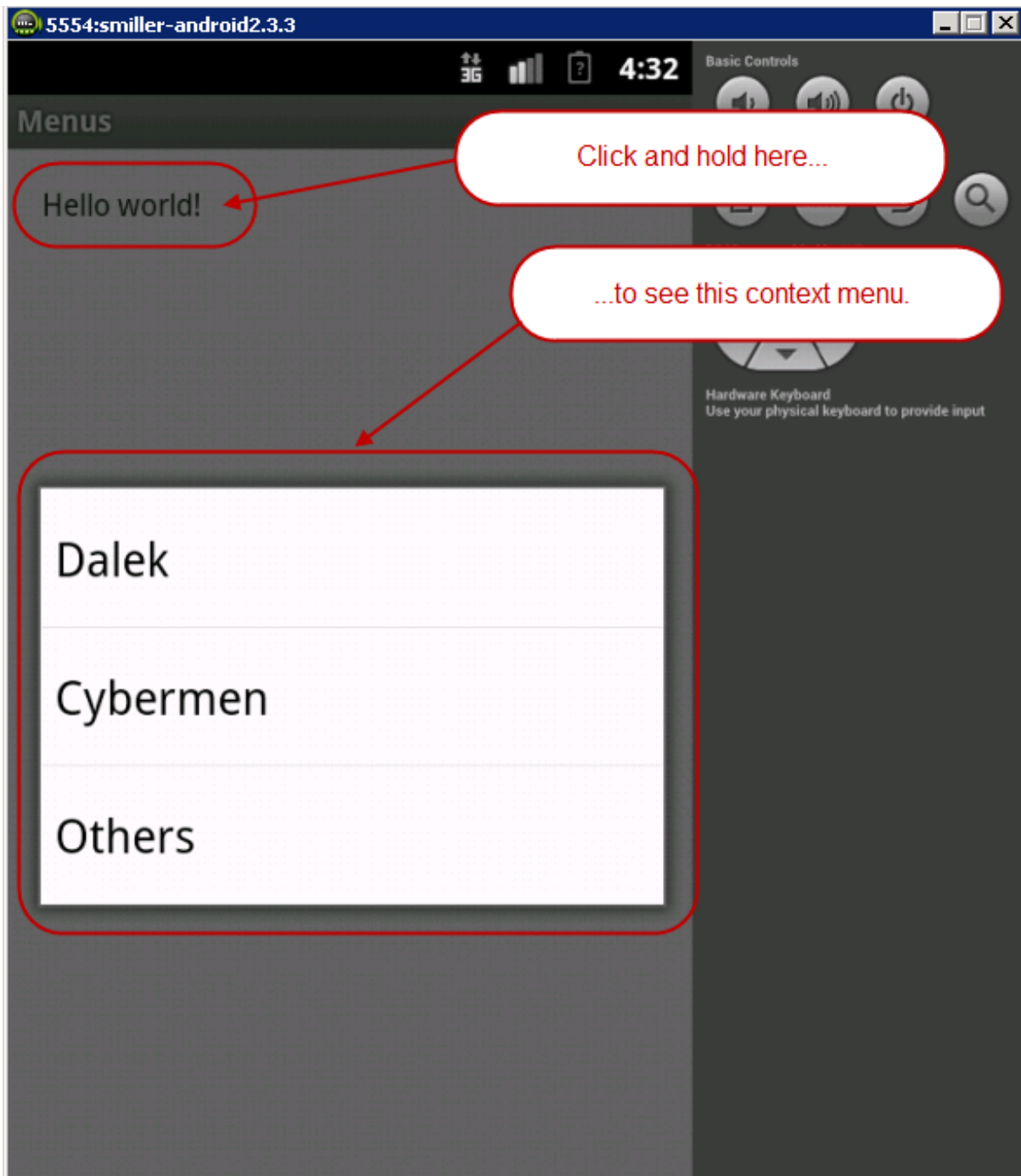
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        optionClickedId = item.getItemId();

        switch (optionClickedId) {
            case R.id.mi_dalek:
                Toast.makeText(this, "Exterminate!", Toast.LENGTH_LONG).show();
                break;
            case R.id.mi_cybermen:
                Toast.makeText(this, "You will be upgraded.", Toast.LENGTH_SHORT).show();
                break;
        }
    }
}
```



```
        break;
    }
    return true;
}
```

In our Activity, we must override and implement the **onCreateContextMenu()** method in order to handle creating the context menu when our registered View has been long-pressed (that is, when part of your screen has been tapped and held down). We inflated the menu here exactly the same way we did for the Options Menu. This works because we've registered only a single view for a Context Menu, but once we register multiple views (or a list), we'll probably need to add some more code to determine which View is requesting a Context Menu, otherwise each View would present the exact same Context Menu. We can save and run our code now to test the menu. To present the menu, you'll need to click and hold on the TextView (the emulator version of a long-press):



Responding to Context Menu clicks is similar to the Options Menu as well. We just need to override a different method. We can reuse the code from **onOptionsItemSelected()** since we're already inflating the same menu:

MainActivity.java

```
package com.oreillyschool.android1.menus;

import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.widget.Toast;
import android.view.ContextMenu;
import android.view.ContextMenu.ContextMenuInfo;
import android.view.View;

public class MainActivity extends Activity {
    private int optionLastClickedId = -1;
    private int optionClickedId = -1;

    /** Called when the activity is first created. */
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        View textView = findViewById(R.id.text);
        registerForContextMenu(textView);
    }

    @Override
    public void onCreateContextMenu(ContextMenu menu, View v, ContextMenuInfo menuInfo) {
        super.onCreateContextMenu(menu, v, menuInfo);
        final MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.main_menu, menu);
    }

    @Override
    public boolean onContextItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case R.id.mi_dalek:
                Toast.makeText(this, "Exterminate!", Toast.LENGTH_LONG).show();
                break;
            case R.id.mi_cybermen:
                Toast.makeText(this, "You will be upgraded.", Toast.LENGTH_SHORT).show();
                break;
        }
        return true;
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        final MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.main_menu, menu);
        return true;
    }

    @Override
    public boolean onPrepareOptionsMenu(Menu menu) {
        MenuItem item = menu.findItem(optionLastClickedId);
        if (item != null) {
            item.setEnabled(true);
        }
        item = menu.findItem(optionClickedId);
        if (item != null) {
            item.setEnabled(false);
        }
        optionLastClickedId = optionClickedId;
    }
}
```

```

        return super.onPrepareOptionsMenu(menu);
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        optionClickedId = item.getItemId();

        switch (optionClickedId) {
            case R.id.mi_dalek:
                Toast.makeText(this, "Exterminate!", Toast.LENGTH_LONG).show();
                break;
            case R.id.mi_cybermen:
                Toast.makeText(this, "You will be upgraded.", Toast.LENGTH_SHORT).show();
                break;
        }
        return true;
    }
}

```

Save and run the app once more to test that the click handler is working correctly. The method for handling Context Menu clicks works exactly the same way as the Options Menu click handler method. However, while the Options Menu always comes from the same menu source, the Context Menu could potentially be generated from any View in the activity that registered to display a Context Menu, so you might need to write some defensive code to determine which View initiated the menu. This is especially true when using a Context Menu with a ListView.

Let's add a ListView to this application to see how to use Context Menu with a list. We can reuse the list code from our earlier lesson that covered the ListView component. If you still have the **Lists** project available, go ahead and copy the **/src/MyListAdapter.java** and **res/layout/my_list_item.xml** files into the corresponding folders in this project, as well as the data and setup that was defined in MainActivity.java and the ListView component from activity_main.xml. Don't just copy those last two files over, though; we want to merge, not replace, the list data with our existing code.

If you don't have the previous code or you just want to re-type it, you can follow the change instructions below. Create a new class file named **MyListAdapter** and make it extend the **android.widget.AdapterView** class. Then modify MyListAdapter.java as shown:

MyListAdapter.java

```
package com.oreillyschool.android1.menus;

import com.oreillyschool.android1.menus.MainActivity.MyData;
import android.content.Context;
import android.graphics.Color;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.TextView;

public class MyListAdapter extends ArrayAdapter<MyData> {

    private LayoutInflater inflater;

    public MyListAdapter(Context context, MyData[] data) {
        super(context, R.layout.my_list_item, data);
        inflater = LayoutInflater.from(context);
    }

    @Override
    public View getView(int position, View convertView, ViewGroup root) {
        View view = convertView;
        if (view == null) {
            view = inflater.inflate(R.layout.my_list_item, null);
        }
        MyData data = getItem(position);

        TextView textView = (TextView) view.findViewById(R.id.text);
        textView.setText(data.name);

        View imageView = view.findViewById(R.id.color);
        int color = data.clicked ? Color.RED : Color.BLUE;
        imageView.setBackgroundColor(color);

        return view;
    }
}
```

Next, create a new XML layout file named **my_list_item.xml** as shown:

/res/layout/my_list_item.xml

```
<?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="match_parent"
    android:orientation="horizontal"
    android:gravity="center_vertical">

    <View
        android:id="@+id/color"
        android:layout_width="10dp"
        android:layout_height="50dp" />

    <TextView
        android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>

</LinearLayout>
```

Next, modify **activity_main.xml** and **MainActivity.java** as shown:

/res/layout/activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context=".MainActivity" >

    <TextView
        android:id="@+id/text"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />

    <ListView
        android:id="@android:id/list"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        />

</LinearLayout>
```

MainActivity.java

```
package com.oreillyschool.android1.menus;

import android.app.Activity;
import android.app.ListActivity;
import android.os.Bundle;
import android.view.ContextMenu;
import android.view.ContextMenu.ContextMenuInfo;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.widget.Toast;
import android.widget.ListView;
import java.util.Comparator;

public class MainActivity extends Activity {
public class MainActivity extends ListActivity {
    private int optionLastClickedId = -1;
    private int optionClickedId = -1;

    public class MyData {
        public String name;
        public boolean clicked;
        public MyData(String name) {
            this.name = name;
            this.clicked = false;
        }
    }

    private MyData[] data = new MyData[] {
        new MyData("Odin"),
        new MyData("Thor"),
        new MyData("Loki"),
        new MyData("Baldr"),
        new MyData("Freyr"),
        new MyData("Heimdallr"),
        new MyData("Ullr"),
        new MyData("Meili"),
        new MyData("Hodr"),
        new MyData("Forseti")
    };

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        View textView = findViewById(R.id.text);
        registerForContextMenu(textView);
        MyListAdapter adapter = new MyListAdapter(this, data);
        adapter.sort(new Comparator<MyData>() {
            @Override
            public int compare(MyData arg0, MyData arg1) {
                return arg0.name.compareTo(arg1.name);
            }
        });

        setListAdapter(adapter);
    }

    @Override
    protected void onItemClick(ListView l, View v, int position, long id) {
        MyListAdapter adapter = (MyListAdapter) getListAdapter();
        MyData item = adapter.getItem(position);
        item.clicked = !item.clicked;
        adapter.notifyDataSetChanged();
    }
}
```

```

    }

    @Override
    public void onCreateContextMenu(ContextMenu menu, View v, ContextMenuInfo menuInfo) {
        super.onCreateContextMenu(menu, v, menuInfo);
        final MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.main_menu, menu);
    }

    @Override
    public boolean onContextItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case R.id.mi_dalek:
                Toast.makeText(this, "Exterminate!", Toast.LENGTH_LONG).show();
                break;
            case R.id.mi_cybermen:
                Toast.makeText(this, "You will be upgraded.", Toast.LENGTH_SHORT).show();
                break;
        }
        return true;
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        final MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.main_menu, menu);
        return true;
    }

    @Override
    public boolean onPrepareOptionsMenu(Menu menu) {
        MenuItem item = menu.findItem(optionLastClickedId);
        if (item != null) {
            item.setEnabled(true);
        }
        item = menu.findItem(optionClickedId);
        if (item != null) {
            item.setEnabled(false);
        }
        optionLastClickedId = optionClickedId;

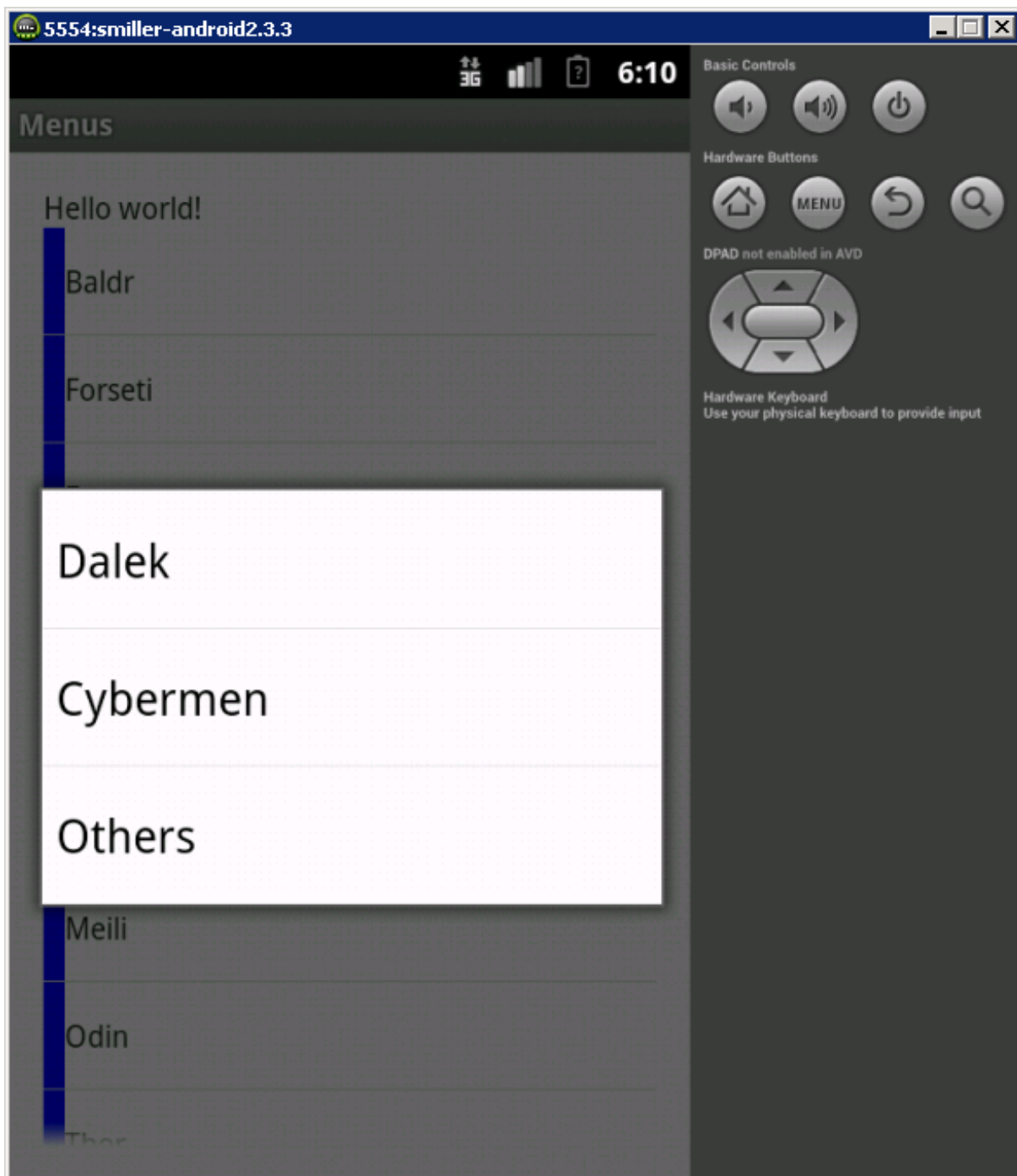
        return super.onPrepareOptionsMenu(menu);
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        optionClickedId = item.getItemId();

        switch (optionClickedId) {
            case R.id.mi_dalek:
                Toast.makeText(this, "Exterminate!", Toast.LENGTH_LONG).show();
                break;
            case R.id.mi_cybermen:
                Toast.makeText(this, "You will be upgraded.", Toast.LENGTH_SHORT).show();
                break;
        }
        return true;
    }
}

```

Before we make any further changes, save everything here and run the project to make sure our previous menus and the list from the previous lesson are working. You'll see the list and still get the Context menu when you click and hold on the TextView:



Now let's make some more changes to get the ListView to show a context menu for each item. We'll start by creating a new menu XML resource. Name the new resource **list_menu.xml** and make these changes:

/res/menu/list_menu.xml

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

    <item
        android:id="@+id/mi_alpha"
        android:title="Alpha" />

    <item
        android:id="@+id/mi_echo"
        android:title="Echo" />

    <item
        android:id="@+id/mi_sierra"
        android:title="Sierra" />

</menu>
```

Now we'll register our list to show a context menu for each item in the list. Make these changes to **MainActivity.java**:

MainActivity.java

```
package com.oreillyschool.android1.menus;

import android.app.ListActivity;
import android.os.Bundle;
import android.view.ContextMenu;
import android.view.ContextMenu.ContextMenuInfo;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.widget.AdapterView.AdapterContextMenuInfo;
import android.widget.ListView;
import android.widget.Toast;
import java.util.Comparator;

public class MainActivity extends ListActivity {
    private int optionLastClickedId = -1;
    private int optionClickedId = -1;

    public class MyData {
        public String name;
        public boolean clicked;
        public MyData(String name) {
            this.name = name;
            this.clicked = false;
        }
    }

    private MyData[] data = new MyData[] {
        new MyData("Odin"),
        new MyData("Thor"),
        new MyData("Loki"),
        new MyData("Baldr"),
        new MyData("Freyr"),
        new MyData("Heimdallr"),
        new MyData("Ullr"),
        new MyData("Meili"),
        new MyData("Hodr"),
        new MyData("Forseti")
    };

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        View textView = findViewById(R.id.text);
        registerForContextMenu(textView);
        registerForContextMenu(getListView());

        MyListAdapter adapter = new MyListAdapter(this, data);
        adapter.sort(new Comparator<MyData>() {
            @Override
            public int compare(MyData arg0, MyData arg1) {
                return arg0.name.compareTo(arg1.name);
            }
        });

        setListAdapter(adapter);
    }

    @Override
    protected void onListItemClick(ListView l, View v, int position, long id) {
        MyListAdapter adapter = (MyListAdapter) getListAdapter();
        MyData item = adapter.getItem(position);
        item.clicked = !item.clicked;
    }
}
```

```

        adapter.notifyDataSetInvalidated();
    }

    @Override
    public void onCreateContextMenu(ContextMenu menu, View v, ContextMenuInfo menuInfo) {
        super.onCreateContextMenu(menu, v, menuInfo);
        final MenuInflater inflater = getMenuInflater();
        if (v == getListView()) {
            inflater.inflate(R.menu.list_menu, menu);
        } else {
            inflater.inflate(R.menu.main_menu, menu);
        }
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        MyData data = null;
        if (item.getMenuInfo() != null && item.getMenuInfo() instanceof AdapterContextMenuItemInfo) {
            AdapterContextMenuItemInfo info = (AdapterContextMenuItemInfo) item.getMenuInfo();
            data = (MyData) getListAdapter().getItem(info.position);
        }

        switch (item.getItemId()) {
            case R.id.mi_dalek:
                Toast.makeText(this, "Exterminate!", Toast.LENGTH_LONG).show();
                break;
            case R.id.mi_cybermen:
                Toast.makeText(this, "You will be upgraded.", Toast.LENGTH_SHORT).show();
                break;
            case R.id.mi_alpha:
            case R.id.mi_echo:
            case R.id.mi_sierra:
                if (data != null)
                    Toast.makeText(this, "You clicked " + item.getTitle(), Toast.LENGTH_LONG).show();
                break;
        }
        return true;
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        final MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.main_menu, menu);
        return true;
    }

    @Override
    public boolean onPrepareOptionsMenu(Menu menu) {
        MenuItem item = menu.findItem(optionLastClickedId);
        if (item != null) {
            item.setEnabled(true);
        }
        item = menu.findItem(optionClickedId);
        if (item != null) {
            item.setEnabled(false);
        }
        optionLastClickedId = optionClickedId;

        return super.onPrepareOptionsMenu(menu);
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {

```

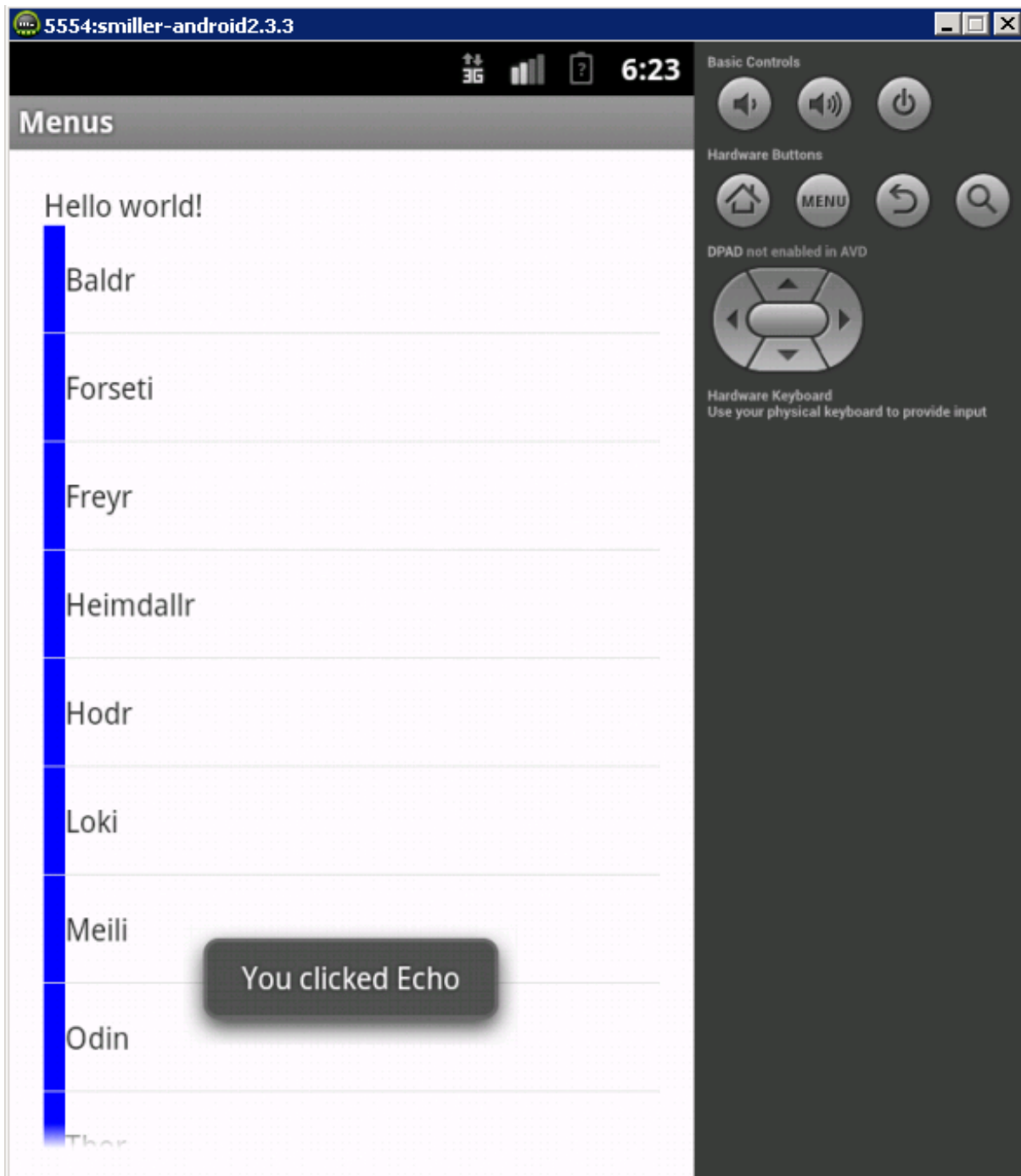
```

        optionClickedId = item.getItemId();

        switch (optionClickedId) {
            case R.id.mi_dalek:
                Toast.makeText(this, "Exterminate!", Toast.LENGTH_LONG).show();
                break;
            case R.id.mi_cybermen:
                Toast.makeText(this, "You will be upgraded.", Toast.LENGTH_SHORT
).show();
                break;
        }
        return true;
    }
}

```

Save your changes and test the app. You'll see the new Context Menu when you long-press on a list item. If you tap a Context menu item, you'll see the corresponding Toast message:



Now we'll make a change to the **onContextItemSelected** method in MainActivity so it looks up which ListItem was clicked and displays both the list item name and the menu option name that was clicked in the Toast message:

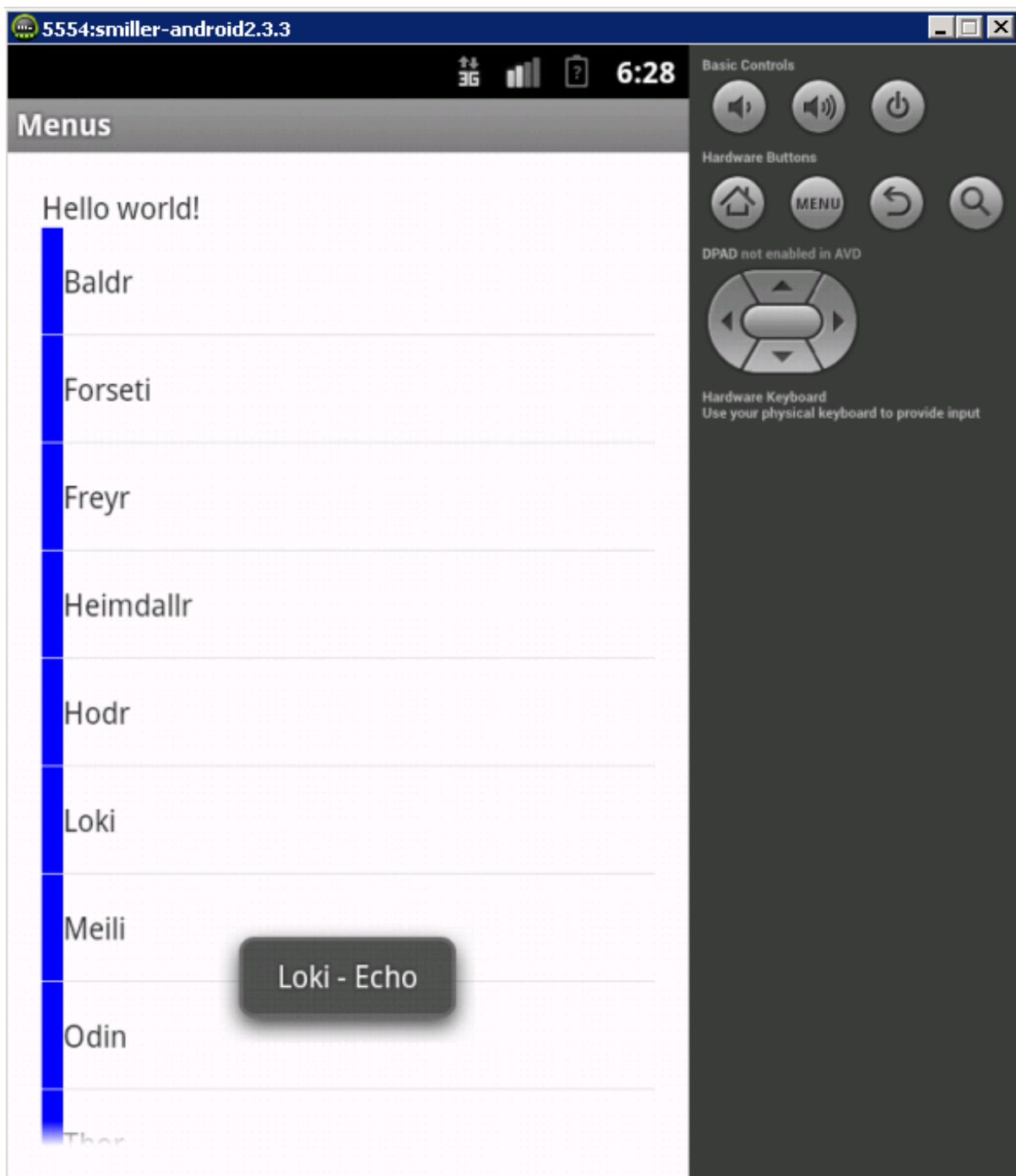
MainActivity.java

```
...
    @Override
    public boolean onContextItemSelected(MenuItem item) {
        MyData data = null;
        if (item.getMenuInfo() != null && item.getMenuInfo() instanceof AdapterC
ontextMenuInfo) {
            AdapterContextMenuInfo info = (AdapterContextMenuInfo) item.getMenuI
nfo();
            data = (MyData) getListAdapter().getItem(info.position);
        }

        switch (item.getItemId()) {
            case R.id.mi_dalek:
                Toast.makeText(this, "Exterminate!", Toast.LENGTH_LONG).show();
                break;
            case R.id.mi_cybermen:
                Toast.makeText(this, "You will be upgraded.", Toast.LENGTH_SHORT
).show();
                break;
            case R.id.mi_alpha:
            case R.id.mi_echo:
            case R.id.mi_sierra:
                if (data != null)
                    Toast.makeText(this, data.name + " - " + item.getTitle(), To
ast.LENGTH_LONG).show();
                break;
        }
        return true;
    }
...

```

Save and run the code. When you long-press an item and then select alpha, echo, or sierra, you'll see a message with both selections.



OBSERVE:

```
MyData data = null;
if (item.getMenuInfo() != null && item.getMenuInfo() instanceof AdapterContextMe
nuInfo) {
    AdapterContextMenuInfo info = (AdapterContextMenuInfo) item.getMenuInfo();
    data = (MyData) getListAdapter().getItem(info.position);
}
```

In the beginning of the method we call `getMenuInfo()` on the `MenuItem` object that was passed to the method. If the source of this Context Menu is a list view item, the `getMenuInfo()` returns an instance of `AdapterContextMenuInfo`. If the source of the Context Menu is a `TextView`, `getMenuInfo()` will only return a null, so we have to write some defensive code here to make sure we don't get a null pointer error. If we get an `AdapterContextMenuInfo` object, we can use it to find the position of the list item, and with that we can retrieve the model that was used to create the list item and then use the model name property in the Toast message.

Note

Unlike the Options Menu, the Context Menu's create method gets called after each long-press on the registered View, so there's no "prepare" method to override in order to make changes to the menu before it is shown; instead, you just put the logic in the **onCreateContextMenu()** method. Keep in mind that Context Menus do not support icons.

Wrapping Up

Well, it seems like menus can pop up anywhere, huh? That's good, because they allow you to provide additional functionality to the screen without cluttering up the view. Now that you have experience creating, modifying, and handling each of the various types of Menus, feel free to play around with these tools on your own. If you feel like you'd like a bit more guidance, hit up the [Android documentation site](#) to explore Menus even further.

When you're ready, move on to the next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Saving Data with Shared Preferences

Good to have you back. I appreciate your persistence! This lesson covers Shared Preferences. Shared Preferences help your application to remember decisions and the state of the data between each application session. There are other, more involved methods of data persistence available on Android (such as a sqlite database), but we'll cover those in future lessons.

Shared Preferences

For this lesson, we'll recycle code from before. There was a lot going on in the code from that last project, but that density will make it especially useful for us to use to test the Shared Preferences feature.

Think of a Shared Preference as a basic data model to which your application can read and write efficiently. The `SharedPreferences` class is the interface used to communicate with the data. You can write any primitive data object to the `SharedPreferences` data model (such as `int`, `float`, `long`, and `string`). The data is saved as a key-value pair.

`SharedPreferences` doesn't support complex data models, but it works well for preserving application state and personalized user settings between sessions. There's actually a specialized Activity for managing user settings that handles much of the work automatically. We'll get into that later, but right now let's go over with the basics of using the `SharedPreferences` class.

Note To save space in this and future lessons, we'll sometimes just show relevant portions of the programs in our CODE TO TYPE boxes. We'll use ellipses (...) to indicate that some code has been omitted.

Getting Started with SharedPreferences

We'll jump right in and integrate `SharedPreferences` into our existing code. Previously, we created an Options Menu that would disable the previously selected option. Let's modify our code to persist the previous disabled option between sessions. In the **Menus** project, open the **MainActivity.java** file, and make these changes:

MainActivity.java

```
...
import android.app.ListActivity;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.view.ContextMenu;
import android.view.ContextMenu.ContextMenuInfo;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.widget.AdapterView.AdapterContextMenuInfo;
import android.widget.ListView;
import android.widget.Toast;
import java.util.Comparator;
...
public class MainActivity extends ListActivity {

    private static final String DISABLED_OPTION_KEY = "disabledOption";
    private int optionLastClickedId = -1;
private int optionClickedId = 1;

    ...
}
```

We start by defining a permanent key to use in the key-value pair. In larger projects, it's sometimes more practical to define constant values in a helper/utility class, but we didn't do that here because we're focused on using `SharedPreferences`. We also delete the `optionClickedId` variable, because we'll be handling that entirely in `SharedPreferences` now. Go ahead and make the next set of changes to **MainActivity.java**:

MainActivity.java

```
...

@Override
public boolean onPrepareOptionsMenu(Menu menu) {
    MenuItem item = menu.findItem(optionLastClickedId);
    if (item != null) {
        item.setEnabled(true);
    }

    int optionClickedId = getPreferences(MODE_PRIVATE).getInt(DISABLED_OPTION_KEY, -1);

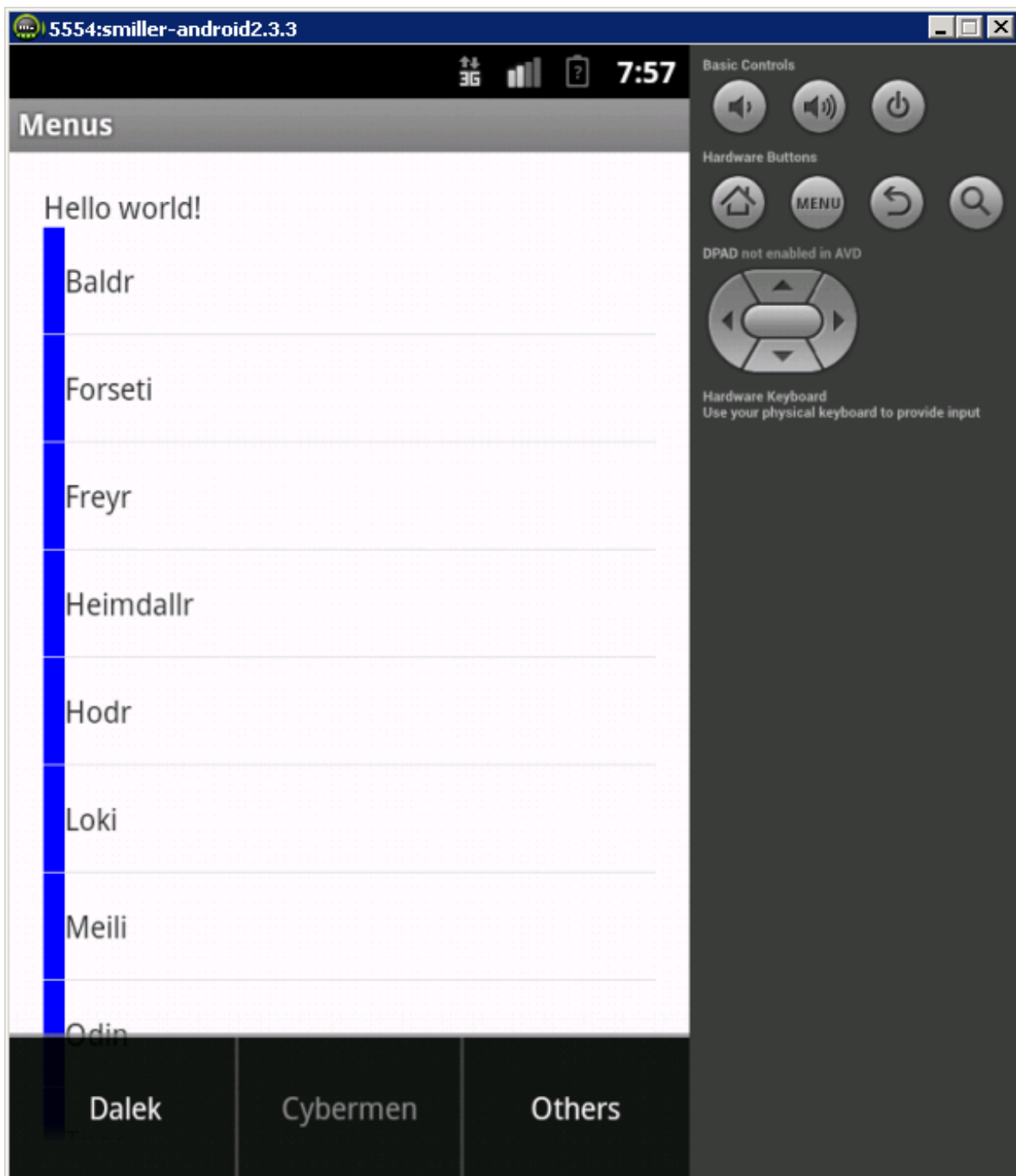
    item = menu.findItem(optionClickedId);
    if (item != null) {
        item.setEnabled(false);
    }
    optionLastClickedId = optionClickedId;

    return super.onPrepareOptionsMenu(menu);
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
optionClickedId = item.getItemId();
    int optionClickedId = item.getItemId();
    final SharedPreferences prefs = getPreferences(MODE_PRIVATE);
    final SharedPreferences.Editor editor = prefs.edit();
    editor.putInt(DISABLED_OPTION_KEY, optionClickedId);
    editor.commit();

    switch (optionClickedId) {
        case R.id.mi_dalek:
            Toast.makeText(this, "Exterminate!", Toast.LENGTH_SHORT).show();
            break;
        case R.id.mi_cyberman:
            Toast.makeText(this, "You will be upgraded.", Toast.LENGTH_SHORT).show();
            break;
    }
    return true;
}
}
```

Save the program. To make sure your code works, run the application, click the Menu button, choose a menu option and make sure that it's disabled when you open the menu again, close the application (hit the **back** button until you see the desktop), and then re-open the application to make sure the menu item that was disabled before, is still disabled. You should be able to restart the emulator and the application should still persist the disabled menu option:



Since we are no longer using the `optionClickedId` class member variable, we have to update each reference that uses it to use the `SharedPreferences` class instead. Take a look at `onPrepareOptionsMenu()`:

OBSERVE:

```
int optionClickedId = getPreferences(MODE_PRIVATE).getInt(DISABLED_OPTION_KEY, -1);
```

The first fix was to load the value out of the `SharedPreferences`. We call the method `getPreferences()` (available from `Activity`) to get an instance of the `SharedPreferences` class unique to the current `Activity`. The method takes **one parameter**, which defines the permissions of the preferences file that is created for the `Activity`. The `Activity` class defines a series of constants to help you configure the privacy correctly: `MODE_PRIVATE`, `MODE_APPEND`, `MODE_WORLD_READABLE`, and `MODE_WORLD_WRITABLE`. We use **MODE_PRIVATE** to keep the preferences private and inaccessible to other languages. If a preference file already exists with a privacy other than `MODE_PRIVATE`, that preference file will be deleted and a new one created. Using `MODE_APPEND` as our parameter would only create a new preference file if one didn't exist, regardless of privacy setting. `MODE_WORLD_READABLE` allows other apps to read the preference file, and `MODE_WORLD_WRITABLE` allows other apps to read and modify the preference file.

We use the `getInt()` method, sending it the `DISABLED_OPTION_KEY` key, to retrieve the saved value for our key-value pair. This method (as well as every other get method on `SharedPreferences`) takes a second parameter as a "default value" to be returned if the key-value pair does not exist yet in the `SharedPreferences` file. It is safe to use `-1` as a default value here because resource id parameters never use negative values:

OBSERVE: `onOptionsItemSelected()`

```
int optionClickedId = item.getItemId();
final SharedPreferences prefs = getPreferences(MODE_PRIVATE);
final SharedPreferences.Editor editor = prefs.edit();
editor.putInt(DISABLED_OPTION_KEY, optionClickedId);
editor.commit();
```

Next, we update `onOptionsItemSelected()` to save the correct value into `SharedPreferences`. Again we use `getPreferences` with the same privacy parameter to get access to our `SharedPreferences` object. To write a value to the preference file, you have to get an instance of the `SharedPreferences.Editor` class by calling the `edit()` method on the `SharedPreferences` class. Then, with the Editor, we use the helper method to update the key-value pair to the preference. Finally, we call `commit()` on the Editor class in order to write the value to the `SharedPreferences` file; if we didn't call `commit()`, `SharedPreferences` would roll back to their previous values.

There are many other helper methods available on the Editor class for the different types of value that are supported by `SharedPreferences`, such as `putString()` and `putLong()`. Instead of using the `getPreferences()` method to get the `SharedPreferences` object, we could have used the `getSharedPreferences()` method. Both methods work almost identically, but `getSharedPreferences()` takes another parameter (a `String`) that is used as a unique name for the `SharedPreferences` file that is created for the Activity. Using `getSharedPreferences()`, you can create as many different `SharedPreferences` files as you want. If you need only a single preference cache for a single Activity, use `getPreferences()`. If you want multiple activities to have access to the same preference class, then you have a few options available. You could use `getSharedPreferences()` and use the same name in each Activity that loads the preference, or you could use the Application class to create a default `SharedPreferences`; for example, `getApplication().getPreferences(MODE_PRIVATE)`.

We haven't covered the Application class in great detail, but there's not typically much need to do that. The Application class functions almost exactly like an Activity with similar methods available to it (both Activity and Application extend the Context class). However, each Android app has only one Application, and it can be accessed from any Activity class using the `getApplication()` method.

Probably the best option for making sure all your activities use the same preferences file is the `PreferenceManager` class. It has a static method called `getDefaultSharedPreferences()` that takes one parameter (a Context such as an Activity or Application) and returns a default `SharedPreferences` object will use the same file, regardless of the Context that is passed to it. This file is best used with the `PreferenceActivity` class because it's the same `SharedPreferences` file used by that class.

Note

`PreferenceManager.getDefaultSharedPreferences()` is actually using the option I described earlier; it's using the same name for the `getSharedPreferences()` call each time. The name it uses is actually a combination of the application package name and the string `"_preferences"` (`context.getPackageName() + "_preferences"`).

PreferenceActivity

Many applications have what's commonly referred to as a "Settings screen." Android has a native custom Activity available, the `PreferenceActivity`, that is meant to assist you in creating a Settings screen designed specifically for your application. The `PreferenceActivity` has its own unique XML format for declaring its view. The `PreferenceActivity` uses the `PreferenceManager` class to manage its `SharedPreferences` object as well, so that the preferences managed on the activity are available to the rest of the activities in the application.

Note

If you browse the Android developer documentation for help implementing the `PreferenceActivity`, you might find it a bit frustrating. As of Android 3.0, all of the previous APIs for setting up a `PreferenceActivity` were deprecated in favor of the new standard using the `PreferenceFragment` class. Unlike other fragments, however, the `PreferenceFragment` is not available in the support library, so we still have to use the deprecated APIs if we want to support devices running an Android version before 3.0.

Let's implement a PreferenceActivity screen in our application now. Start by creating a new class called **MyPreferenceActivity**; make sure it extends the **PreferenceActivity** class:

The screenshot shows the 'New Java Class' dialog box. The 'Source folder' is set to 'Menus/src', the 'Package' is 'com.ost.android.menus', the 'Name' is 'MyPreferenceActivity', and the 'Superclass' is 'android.preference.PreferenceActivity'. The 'Inherited abstract methods' checkbox is checked. The 'Finish' button is highlighted.

Click **Finish** to create the activity. We have relatively little code to modify for this activity; just make this change:

```
MyPreferenceActivity.java

package com.oreillyschool.android1.menus;

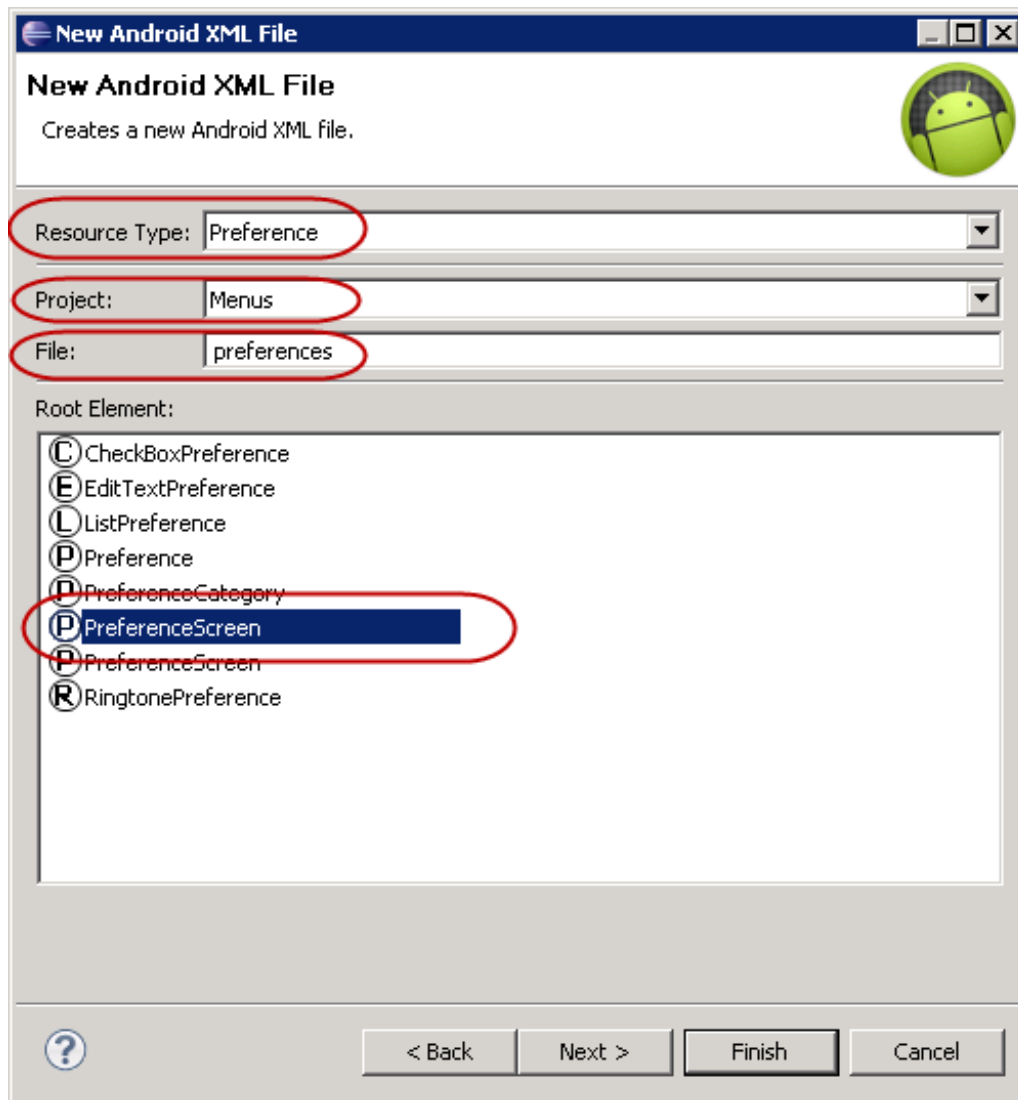
import android.os.Bundle;
import android.preference.PreferenceActivity;

public class MyPreferenceActivity extends PreferenceActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.preferences);
    }
}
```

This should be throwing an error right now since we reference a resources file that we haven't created yet—so let's create it now:

1. With the **Menus** project selected, select **File | New | Other** (or use the keyboard shortcut **Ctrl-N**).
2. In the "Select a Wizard" dialog, choose the **Android XML File** option in the **Android** folder, and click **Next**.
3. In the "New Android XML File Wizard," change the "Resource Type" to **Preference**; name the file **preferences**; under "Root Element," select the **PreferenceScreen**; and click **Finish** to create the XML resource.



This creates the **/res/xml** folder in your project (if it doesn't already exist), then creates the new **preferences.xml** file and saves it in there. Now let's add a preference to this XML. Modify your code as shown:

/res/xml/preferences.xml

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

    <PreferenceCategory android:title="Important Stuff">

        <CheckBoxPreference
            android:key="listViewVisible"
            android:title="ListView visible"
            android:defaultValue="true"
            />

        <EditTextPreference
            android:key="username"
            android:title="Username"
            android:defaultValue="User"
            />

    </PreferenceCategory>

    <PreferenceCategory android:title="Unimportant Stuff">

        <CheckBoxPreference
            android:key="doesNothing"
            android:title="Unimportant Text"
            />

    </PreferenceCategory>

</PreferenceScreen>
```

The PreferenceActivity XML supports many different types of standard preference screen components including checkboxes, editable, text areas, lists, as well as preference groups to help organize your preferences. Here we've used two types of components: the **CheckBoxPreference** and the **EditTextPreference**, which will correspond to a CheckBox and EditText view component, respectively.

Before we can even test our code, we need to update **MainActivity.java** with a hook to load this activity. Let's go the quick-and-dirty route of setting up a quick click-listener on the top TextView. Modify your code as shown:

MainActivity.java

```
...
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    View textView = findViewById(R.id.text);
    textView.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            startActivity(new Intent(MainActivity.this, MyPreferenceActivity
.class));
        }
    });
    registerForContextMenu(textView);
registerForContextMenu(getListView());

    ...
}
```

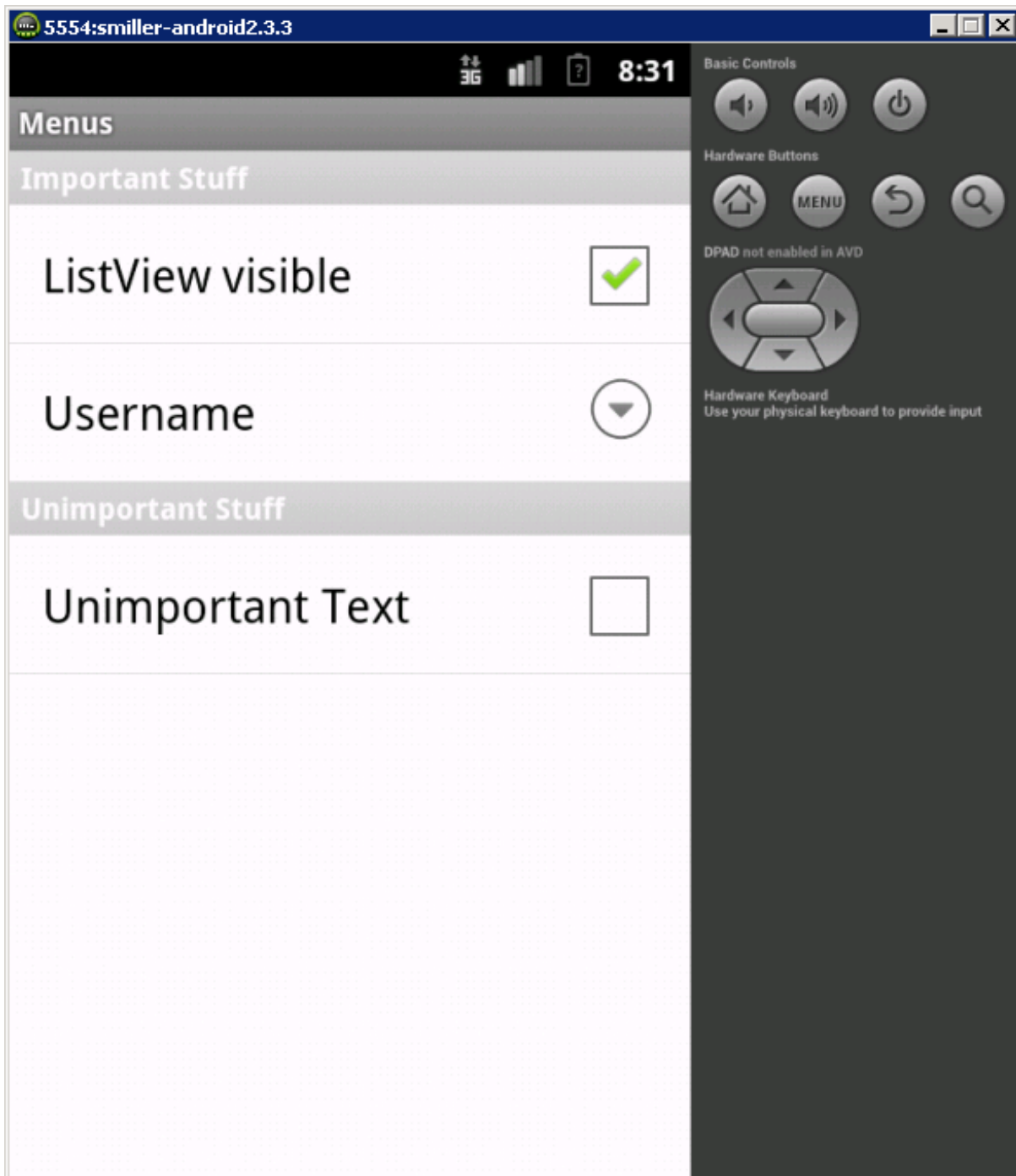
Finally, we need to add the Activity to the **AndroidManifest.xml** file. Modify your code so it looks like this:

AndroidManifest.xml

```
...
    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:label="@string/app_name"
            android:name=".MainActivity" >
            <intent-filter >
                <action android:name="android.intent.action.MAIN" />

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

Now we're ready to test the code. The preferences aren't hooked up to anything yet, but we should at least be able to test to make sure that the `PreferenceActivity` is creating its view correctly. Start up the application, and click the **Hello world** text at the top of the screen; your `PreferenceActivity` screen will look like this:



Note

The **PreferenceScreen** xml tag can even be nested inside of itself. When clicked, this will create an item on the screen that will load a brand new preference screen, populated with the preferences that are children of the nested tag. This is commonly used for things like "Advanced Settings" options in a typical settings screen.

Now let's hook up some of these preferences to verify that they're working. Make these changes to **MainActivity.java**:

MainActivity.java

```
...
@Override
protected void onResume() {
    super.onResume();

    SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences(
this);
    TextView textView = (TextView)findViewById(R.id.text);
    textView.setText(String.format("Hello %s, welcome back!", prefs.getStrin
g("username", "user")));

    if (prefs.getBoolean("listViewVisible", true)) {
        getListView().setVisibility(View.VISIBLE);
    } else {
        getListView().setVisibility(View.INVISIBLE);
    }
}
```

Here we choose to override the **onResume** method instead of using **onCreate**. In the lifecycle of an Android Activity class, the **onCreate** method will get called only once when the Activity is initially prepared and created. The **onResume** method, however, always gets called just before the activity becomes visible. Since the **MainActivity** class isn't destroyed when we load the **MyPreferenceActivity** (just added to the back-stack), we have to manage the update of our preference changes in the **onResume** method. To learn more about the lifecycle of an Activity, I *highly* recommend checking out the [Android developer site](#) (complete with a handy info-graphic).

After loading the **SharedPreferences** object from the **PreferenceManager**, the rest of the code works exactly the same way it did before, using the various "get" methods to retrieve the cached data. Be sure to test your application and verify that the settings page now controls the "name" that's used in the TextView, as well as the CheckBoxPreference controlling whether the ListView is visible.

Note

There are a couple of down-sides to be aware of with the PreferenceActivity. First, you can't use a "static final" variable as your key in the XML preference resource, so make certain that your key strings are always identical (or consider using a strings.xml resource). Also, frequently you'll need to define a default value in multiple areas. It's usually best to keep them consistent. In any case, the benefits of using the **PreferenceActivity** far outweigh these minor inconveniences.

Wrapping Up

We've only just scratched the surface of managing data on Android with these essential classes. Make sure you're confident using a simple SharedPreferences object and setting up a PreferencesActivity. These convenient classes will help you to implement simple data persistence quickly, in any application.

Good work so far. Let's press on!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Saving Data with a Database

Welcome back! In this lesson, we'll work on storing and retrieving data from a database in Android. The Android SDK has built-in classes to help create and manage an SQLite database.

SQLite

Note

While it is not necessary to be an expert in SQLite for this lesson, it's good to have a basic understanding of how databases work, and how to perform SQL queries. If you think you could use some help when it comes to working with databases, consider taking the O'Reilly School of Technology course [PHP/SQL 1: Introduction to Database Programming](#).

Creating a Helper

Let's get started! Create a project named **Database**, name the package **com.oreillyschool.android1.database**, and assign it to the **Android1_Lessons** working set.

The primary class for interacting with a SQLite database in Android is the **SQLiteOpenHelper** class. The **SQLiteOpenHelper** class is an abstract class to be used for creation and version management of the SQLite database. Let's set up a basic **SQLiteOpenHelper** implementation first. In the **Database** project, create a new class file named **DBHelper** that extends the **SQLiteOpenHelper** class:

New Java Class

Java Class
Create a new Java class.

Source folder: Database/src Browse...

Package: com.ost.android1.database Browse...

☐ Enclosing type: Browse...

Name: DBHelper

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass: android.database.sqlite.SQLiteOpenHelper Browse...

Interfaces: Add... Remove

Which method stubs would you like to create?

☐ public static void main(String[] args)
☐ Constructors from superclass
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))
☐ Generate comments

Finish Cancel

SQLiteOpenHelper has two abstract methods that we must implement: **onCreate()** and **onUpgrade()**. Each method is intended to be used to modify the structure of a database after the respective event. During **onCreate**, we'll set up the basic tables for all the data, and in **onUpgrade**, we'll enter any migration logic needed to convert a database from an older version to match the new database. Make sure that both methods result in the database having the same schema, regardless of whether they just installed the application or are updating from a previous application version.

You might have noticed that the class generated from the "New Class Wizard" throws a compiler error initially. This is because we haven't implemented a constructor for the class to complete our implementation. Let's tackle that now. Make these changes to your code:

DBHelper.java

```
package com.oreillyschool.android1.database;

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

public class DBHelper extends SQLiteOpenHelper {

    private static final String DB_NAME = "myDatabase.db";
    private static final int DB_VERSION = 1;

    public DBHelper(Context context) {
        super(context, DB_NAME, null, DB_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        // TODO Auto-generated method stub
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // TODO Auto-generated method stub
    }

}
```

In the constructor, we call the super constructor with the appropriate values:

OBSERVE:

```
private static final String DB_NAME = "myDatabase.db";
private static final int DB_VERSION = 1;

public DBHelper(Context context) {
    super(context, DB_NAME, null, DB_VERSION);
}
```

We can predefine the last three values from within this class, but we'll need a **Context** from whatever class is attempting to access the database (most likely an Activity). The second parameter is the **unique name** to use for our database. This should never change, especially when you update the application. If you need multiple databases, make sure the name value is unique for each database. We can safely ignore the **third parameter** for now. The last parameter is the **version of the Database** that we are currently using. Any time you update the application and you have to make changes to the schema of your database, you should increment this version id (just change the "static final" variable). If the **SQLiteOpenHelper** detects an existing database with a version lower than this id, the **onUpgrade()** method will be called instead of **onCreate()**.

Let's define and initialize a simple database for use in our application in the **onCreate()** method:

DBHelper.java

```
...

public class DBHelper extends SQLiteOpenHelper {

    private static final String DB_NAME = "myDatabase.db";
    private static final int DB_VERSION = 1;

    public static final String TABLE_PEOPLE = "people";
    public static final String C_ID = "_id";
    public static final String C_NAME = "name";

    public DBHelper(Context context) {
        super(context, DB_NAME, null, DB_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        // TODO Auto-generated method stub
        final String sqlCreateTablePeople = "CREATE TABLE "
            + TABLE_PEOPLE + " ( " + C_ID
            + " integer primary key autoincrement, " + C_NAME
            + " text not null);";
        db.execSQL(sqlCreateTablePeople);
    }

    ...
}
```

We added a few more "static final" String values to the class that defines the table and column names. We left these public to allow other classes to use them as well. In the **onCreate()** method, we created the SQL statement necessary to add our table to the database using a String, and executed the string by passing it to the **SQLiteDatabase** method **execSQL()**. The **execSQL()** statement is used only for quick SQL commands to execute on the database when you don't require any feedback from the database. This makes it ideal for schema updates such as creating/deleting a table or modifying table columns.

Next, let's write a quick and basic implementation of the **onUpgrade()** method:

DBHelper.java

```
...
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // TODO Auto-generated method stub
        final String sqlDropTablePeople = "DROP TABLE IF EXISTS " + TABLE_PEOPLE
        + ";";
        db.execSQL(sqlDropTablePeople);
        onCreate(db);
    }
}
```

For **onUpgrade()**, we drop the "people" table (if it already exists) and then recreate it by calling the **onCreate()** method. This is the easiest way to safely implement a database upgrade, but with one glaring potential concern: depending on your application, you might want to preserve any data already in the database during an upgrade. In that situation you would need to write the appropriate migration logic for your tables to correctly update the database schema.

Using the Helper

Now that we've created a simple helper and defined our database, we'll need to write some code to use this data in a view. Let's start with updating the **activity_main.xml** layout file. Modify your code as shown:

/res/layout/activity_main.xml

```
<RelativeLinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context=".MainActivity" >

    <del>TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />
    </del>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <Button
            android:id="@+id/add_btn"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Add Person"
            android:onClick="onAddClicked" />

        <Button
            android:id="@+id/delete_btn"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Delete Person"
            android:onClick="onDeleteClicked" />

    </LinearLayout>

    <ListView
        android:id="@android:id/list"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</RelativeLinearLayout>
```

Here we have two buttons at the top: one to add rows to the database and one to remove rows. We'll also use a ListView to display the data in the database. Next, let's create a view for a dialog that can be present when we click the **New Person** button. Create a new Android Layout XML file named **add_person_dialog** and make these changes:

/res/layout/add_person_dialog.xml

```
<?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="match_parent"
    android:orientation="vertical" >

    <EditText
        android:id="@+id/name"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Name" />

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >

        <Button
            android:id="@+id/okay_btn"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Okay" />

        <Button
            android:id="@+id/cancel_btn"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Cancel" />

    </LinearLayout>
</LinearLayout>
```

This basic view with an EditText and two Buttons should work. Next, modify **MainActivity.java** to hook the dialog up to the "Add Person" button and implement an insert on our database:

MainActivity.java

```
package com.oreillyschool.android1.database;

import android.app.Activity;
import android.app.Dialog;
import android.content.ContentValues;
import android.database.sqlite.SQLiteDatabase;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;

import android.view.Menu;

public class MainActivity extends Activity {

    private static final int ADD_DIALOG = 0;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void onAddClicked(View view) {
        showDialog(ADD_DIALOG);
    }

    public void onDeleteClicked(View view) {
    }

    @Override
    protected Dialog onCreateDialog(int id) {
        Dialog d;
        switch (id) {
            case ADD_DIALOG:
                d = new Dialog(this);
                d.setContentView(R.layout.add_person_dialog);
                d.setTitle("Add a Person");
                final TextView nameText = (TextView) d.findViewById(R.id.name);
                d.findViewById(R.id.okay_btn).setOnClickListener(new View.OnClickListener() {
                    @Override
                    public void onClick(View v) {
                        addPerson(nameText.getText().toString());
                        dismissDialog(MainActivity.ADD_DIALOG);
                    }
                });
                d.findViewById(R.id.cancel_btn).setOnClickListener(new View.OnClickListener() {
                    @Override
                    public void onClick(View v) {
                        dismissDialog(MainActivity.ADD_DIALOG);
                    }
                });
                break;
            default:
                d = super.onCreateDialog(id);
                break;
        }
        return d;
    }

    public void addPerson(String name) {
        // add the new data to the db
        DBHelper helper = new DBHelper(this);
        SQLiteDatabase db = helper.getWritableDatabase();
        ContentValues cv = new ContentValues();
        cv.put(DBHelper.C_NAME, name);
    }
}
```

```

        db.insert(DBHelper.TABLE_PEOPLE, null, cv);
        db.close();
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }
}

```

The important new code to take note of here is in the new **addPerson()** method:

OBSERVE:

```

public void addPerson(String name) {
    // add the new data to the db
    DBHelper helper = new DBHelper(this);
    SQLiteDatabase db = helper.getWritableDatabase();
    ContentValues cv = new ContentValues();
    cv.put(DBHelper.C_NAME, name);
    db.insert(DBHelper.TABLE_PEOPLE, null, cv);
    db.close();
}

```

Here we use our **DBHelper** class to get a **SQLiteDatabase** object on which we can perform inserts. We also need to create a **ContentValues** object for the new data. The **ContentValues** class holds all the data in a key-value map where the key is the database column to use for the value. Then we use the **SQLiteDatabase insert()** method to commit the data. Its parameters are **the table name**, a "null column hack" string (which we can safely ignore), and the **content values** to be inserted. SQLite only supports a single row insert at a time, so for each insert you want to perform, you must call **insert()** again. Finally, since we are done using the database, we call **close()** on the **SQLiteDatabase**.

At this point we are able to save and test to make sure that our Dialog is being created and dismissed correctly, but we'll still have no idea whether our Database inserts are actually working. For that, we'll have to write some logic to query the database and update the ListView with the results.

Cursor and CursorAdapter

Android provides a wrapper class for retrieving the results from a query to a database called a *Cursor*. Android also provides a convenient implementation of the **Adapter** interface for supplying data to a list through a **Cursor**, called a **SimpleCursorAdapter**. These classes are perfect for testing the results of our earlier code. Make these changes to **MainActivity.java**:

MainActivity.java

```
package com.oreillyschool.android1.database;

import android.app.Activity;
import android.app.Dialog;
import android.app.ListActivity;
import android.content.ContentValues;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.os.Bundle;
import android.view.View;
import android.widget.SimpleCursorAdapter;
import android.widget.TextView;

public class MainActivity extends ActivityListActivity {

    private static final int ADD_DIALOG = 0;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // initialize the adapter
        SimpleCursorAdapter adapter = new SimpleCursorAdapter(this,
            android.R.layout.simple_list_item_single_choice,
            null,
            new String[]{DBHelper.C_NAME},
            new int[]{android.R.id.text1});
        setListAdapter(adapter);

        updateAdapterData();
    }

    public void updateAdapterData() {
        // re-query the data
        SQLiteDatabase db = new DBHelper(this).getReadableDatabase();
        Cursor c = db.query(DBHelper.TABLE_PEOPLE,
            null, null, null, null, null, null);
        ((SimpleCursorAdapter)getListAdapter()).changeCursor(c);
        db.close();
    }

    ...

    public void addPerson(String name) {
        // add the new data to the db
        DBHelper helper = new DBHelper(this);
        SQLiteDatabase db = helper.getWritableDatabase();
        ContentValues cv = new ContentValues();
        cv.put(DBHelper.C_NAME, name);
        db.insert(DBHelper.TABLE_PEOPLE, null, cv);
        db.close();

        // update the view
        updateAdapterData();
    }
}
```

Now we should be able to save and run the code to make sure that our add method is correctly adding to the database, as well as see our data being loaded correctly into the **List View**:



There's a lot going on in here. Let's look at it bit by bit:

OBSERVE: onCreate()

```
// initialize the adapter
SimpleCursorAdapter adapter = new SimpleCursorAdapter(this,
    android.R.layout.simple_list_item_1,
    null,
    new String[]{DBHelper.C_NAME},
    new int[]{android.R.id.text1});
setListAdapter(adapter);
```

We start off by creating **an instance of the SimpleCursorAdapter** for our list, which takes five parameters for its constructor: a **Context**, an **int layout id reference**, a **Cursor**, a **String array of column names**, and an **int array of view ids**. For the **layout**, we use a default layout available to us in Android which provides a basic **TextView**. For the **Cursor**, we send null for now, because we're handling that later in the **updateAdapterData()** method. The last two array parameters are intended to match one another in length, so that the values from each column defined in the **String** array will be assigned to the respective

view component with the id defined in the `int` array.

After creating this **adapter**, we assign it to the list through the **ListActivity** method `setListAdapter()`. And finally we call the **updateAdapterData** method that we just created below to load the data into the list. We must refresh the **Cursor** after each addition to keep that logic in a helper method so we're not writing the same code in multiple areas:

OBSERVE: `updateAdapterData()`

```
public void updateAdapterData() {
    // re-query the data
    SQLiteDatabase db = new DBHelper(this).getReadableDatabase();
    Cursor c = db.query(DBHelper.TABLE_PEOPLE,
        null, null, null, null, null, DBHelper.C_NAME);
    startManagingCursor(c);
    ((SimpleCursorAdapter) getListAdapter()).changeCursor(c);
    db.close();
}
```

In `updateAdapterData()`, we use the **DBHelper** class again. This time we call the `query()` method to retrieve the data. The `query()` method is a helper method for performing a "SELECT" query on the database. It takes many parameters in order to support many various types of "SELECT" queries. We are performing only basic queries, so we end up passing null to many of the parameters. The first parameter is the **table name**—it is not optional. The only other parameter we're sending is the last one, which defines **by which column the results should be sorted**. If we didn't care about sorting, we could pass **null** for that parameter as well. This `query()` call is the equivalent to the SQL "SELECT * FROM people ORDER BY name;". Proper usage of the `query()` method sanitizes the query automatically, to prevent SQL injection hacks.

We won't go into all the parameters available to `query()` here, but most of them are more or less self-explanatory when you read the code hints in Eclipse. If you want to read more about the parameters, check the Android developer documentation on the [SQLiteDatabase query method](#).

Finally, we'll implement the "delete" button that we created earlier. Make these changes to **MainActivity.java**:

MainActivity.java

```
...
import android.widget.ListView;
...

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        final ListView list = getListView();
        list.setItemsCanFocus(false);
        list.setChoiceMode(ListView.CHOICE_MODE_SINGLE);

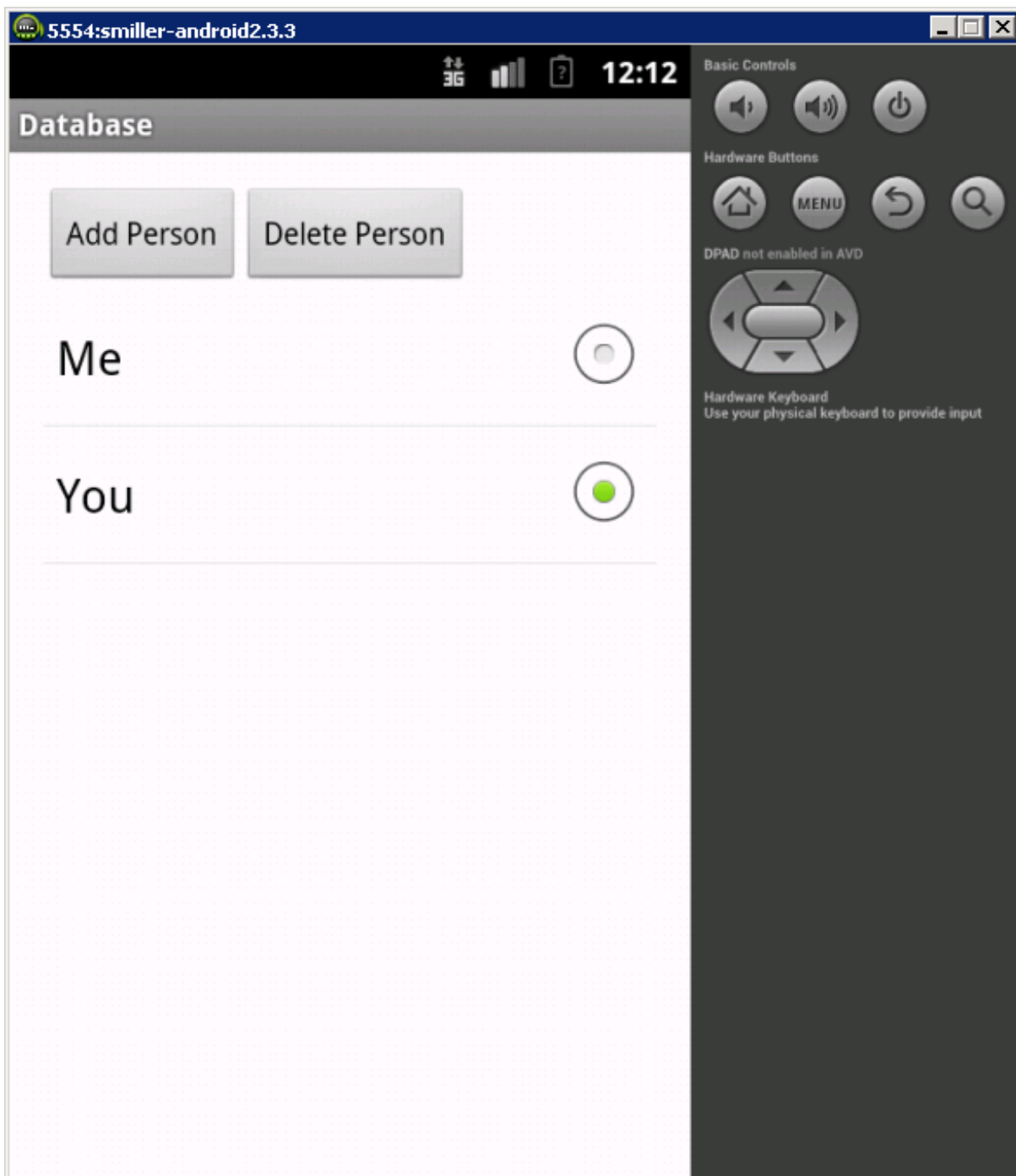
        // initialize the adapter
        SimpleCursorAdapter adapter = new SimpleCursorAdapter(this,
            android.R.layout.simple_list_item_single_choice,
            null,
            new String[]{DBHelper.C_NAME},
            new int[]{android.R.id.text1});
        setListAdapter(adapter);

        updateAdapterData();
    }

    ...

    public void onDeleteClicked(View view) {
        int position = getListView().getCheckedItemPosition();
        if (position >= 0) {
            long itemId = getListAdapter().getItemId(position);
            SQLiteDatabase db = new DBHelper(this).getWritableDatabase();
            int rowsAffected = db.delete(DBHelper.TABLE_PEOPLE, DBHelper.C_ID +
" = " + itemId, null);
            db.close();
            if (rowsAffected > 0)
                updateAdapterData();
        }
    }
}
```

Now you'll be able to run the application again and delete whatever row is checked:



We made a few changes to the list here to allow us to select an individual row. The default layout **simple_list_item_single_choice** looks just like the previous layout, but with a **RadioButton** added to the side of the row as well. By calling **setChoiceMode()** on the list earlier and giving it the parameter **ListView.CHOICE_MODE_SINGLE**, we instruct the list to manage the checked row, and to allow only one row to be checked at a time.

In the **onDeleteClicked()** method, we implemented the delete action. We use the **ListView** method **getCheckedItemPosition()** to find out which item is checked. This could potentially be -1 if no item is checked, so we code defensively around that. Then we have to use the **ListAdapter** to retrieve the actual row id of the item. This corresponds to the **"_id"** column of the data in the database. Finally, we get a writable version of the database from our helper again, only this time we call **delete**, giving it the table name, and an SQL **"WHERE"** clause to delete just the row that matches the **"_id"** of the list item that is checked. Passing null to the second argument would delete all rows in the table. The final argument is used to help sanitize the query again by replacing any question marks in the **"WHERE"** clause with a sanitized value from the third argument. We aren't concerned with that here, so we just pass in null. If any rows were affected by our delete query, then we update the adapter (after closing the database, of course).

Wrapping Up

Properly managing a database in Android can seem like quite a daunting task. Thankfully Android has provided many convenient helper classes to make that easier. Hopefully by now you're feeling comfortable with performing the CRUD actions (create, read, update, delete) on a database in Android. There's a lot of advanced SQLite content that we didn't cover in this lesson, but this is an excellent foundation and usually enough for most applications. If you find yourself in need of some really advanced SQLite work in your application, as always, be sure to check out the [Android developer documentation site](#).

See you in the next lesson!

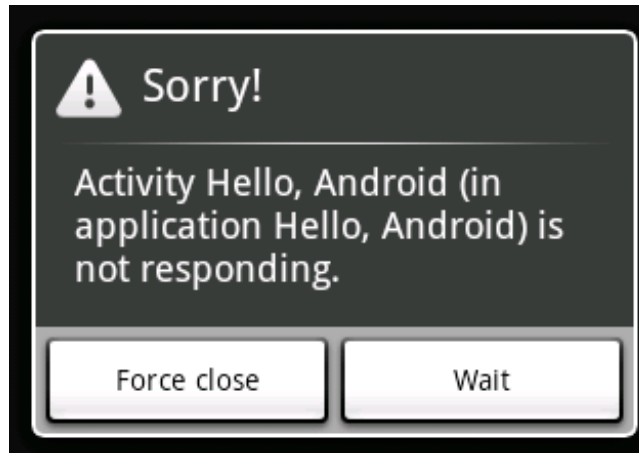
Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Threading with AsyncTasks

Welcome back! This lesson covers threading in Android. Implementing proper threading is crucial for Android application development. It helps you to maintain fast and seamless views that enable a good user experience, and even more importantly, it can help prevent your application from falling into an Application Not Responding (ANR) state.



An ANR Dialog will be presented over your application in the event that the user interface doesn't respond to input events (such as a screen touch or key press event) within 5 seconds. When the ANR dialogue is presented, the Android system (and the user) assume that your application has crashed and will not recover. In most cases, this will lead to negative reviews on the Android app market as well.

Threading in Android

In Android, every Application is assigned a default "main" thread, commonly referred to as the UI Thread. All work for an application is done on this thread by default, including user interface drawing, event dispatching and handling, and all the code that you write. If you write code that takes a long time to finish, during that time, your interface may not be able to draw updates to the screen. This is often the cause of an ANR state. To prevent an occurrence of an ANR state, we'll create a separate thread to handle the work that, when executed, will run asynchronously. The Android SDK provides a helper class for creating and managing work in a separate Thread, called an **AsyncTask**.

AsyncTask

AsyncTask is a helper class that makes it easier to spin off a Thread to do work, track progress, and respond to the results. It can be a little confusing to set up at first because it takes three generic parameters. We'll set up a new project named **Threads**, with package name **com.oreillyschool.android1.threads**, assigned to the **Android1_Lessons** working set.

Let's create a short method that fakes some heavy work. Make these changes to **MainActivity.java**:

MainActivity.java

```
package com.oreillyschool.android1.threads;

import android.app.Activity;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.os.Bundle;
import android.view.Menu;

public class MainActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    private Bitmap downloadImage() {
        final long start = System.currentTimeMillis();

        // wait 5 seconds (5000 milliseconds) until proceeding
        while (System.currentTimeMillis() - start < 5000) {
        }

        return BitmapFactory.decodeResource(getResources(), R.drawable.ic_launcher);
    }

    @Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.main, menu);
    return true;
}

}
```

This short method will wait five seconds and then return the app icon image as a Bitmap. Next, add some components to help demonstrate our concepts. Make these changes to **activity_main.xml**:

/res/layout/activity_main.xml

```
<RelativeLinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context=".MainActivity" >

    <del>TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="onLoadImageClicked"
        android:text="Load Image" />

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center" >

        <ProgressBar
            android:id="@+id/progress"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:indeterminate="true" />

        <ImageView
            android:id="@+id/image"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:scaleType="center"
            android:visibility="gone" />

    </RelativeLayout>

</RelativeLinearLayout>
```

We added a button to trigger our loading process, as well as a **ProgressBar** and an **ImageView** nested and centered in a **RelativeLayout**. The **ProgressBar** is set to **indeterminate** so that it will spin continuously. The **ImageView** is not initially visible. Once we load the image, we'll hide the **ProgressBar** and show the **ImageView**. Now let's head back to **MainActivity.java** and connect everything:

MainActivity.java

```
package com.oreillyschool.android1.threads;

import android.app.Activity;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.os.Bundle;
import android.view.View;
import android.widget.ImageView;
import android.widget.ProgressBar;

public class MainActivity extends Activity {

    private ImageView image;
    private ProgressBar progress;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        progress = (ProgressBar) findViewById(R.id.progress);
        image = (ImageView) findViewById(R.id.image);
    }

    public void onLoadImageClicked(View view) {
        // show the progress and hide the image
        image.setVisibility(View.GONE);
        progress.setVisibility(View.VISIBLE);

        image.setImageBitmap(downloadImage());

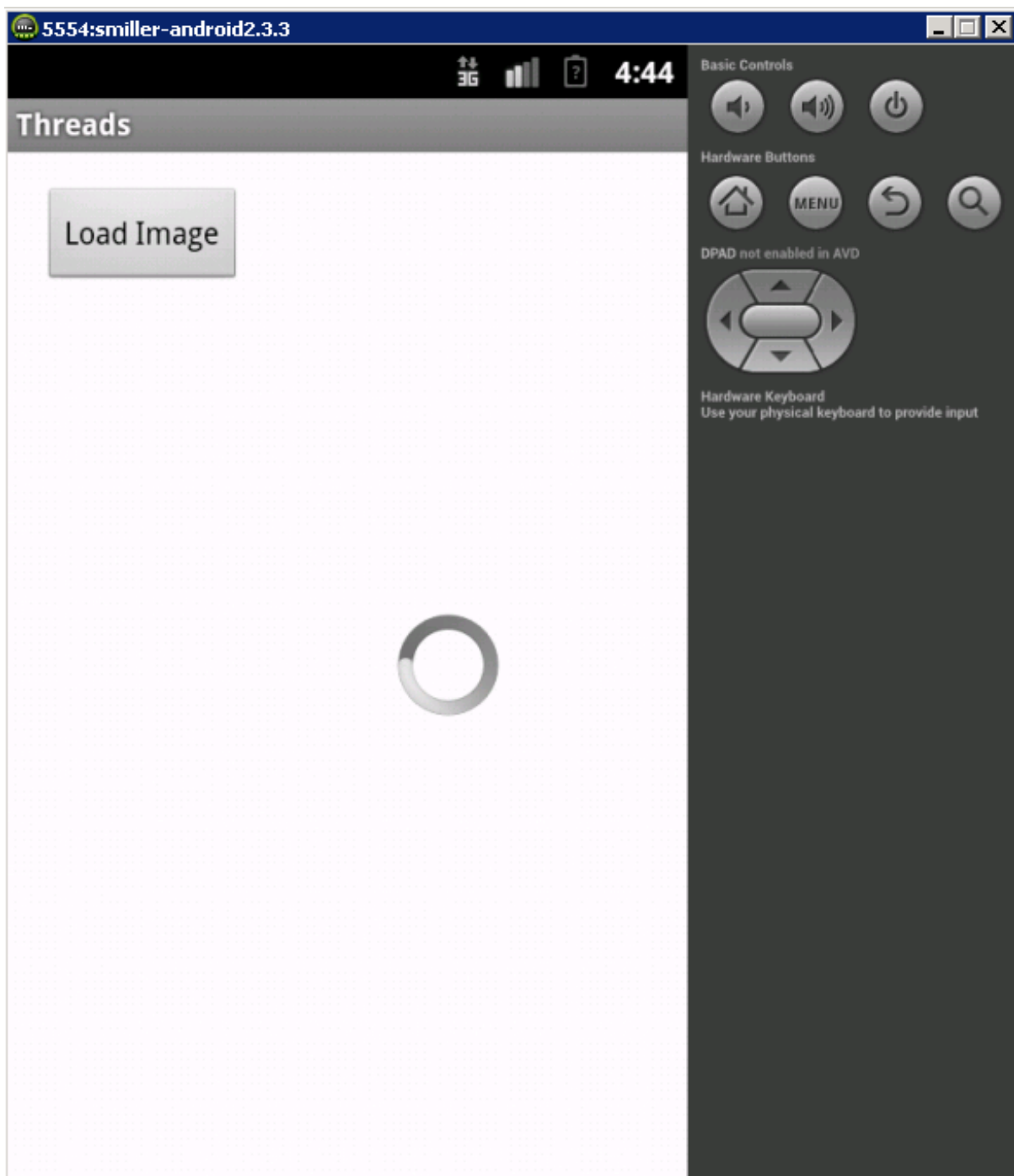
        // show the image and hide the progress
        image.setVisibility(View.VISIBLE);
        progress.setVisibility(View.GONE);
    }

    private Bitmap downloadImage() {
        final long start = System.currentTimeMillis();

        // wait 5 seconds (5000 milliseconds) until proceeding
        while (System.currentTimeMillis() - start < 5000) {
        }

        return BitmapFactory.decodeResource(getResources(), R.drawable.ic_launcher);
    }
}
```

Now that we've hooked everything up, we can give the code a test run. We aren't using an AsyncTask yet, because I want to demonstrate some problems you can run into if you don't use a separate thread to run heavy code. After the application is installed and running, you'll see the infinite progress bar spinning in the middle. If you click **Load Image**, the progress bar spinner will freeze during the five seconds that the image takes to load. Odds are that the button itself will be frozen in the pressed state as well. This would be pretty frustrating to a user to say the least!



Alright, so now that we have code that's freezing the application, let's go back and fix it with a proper implementation of threading. Make these changes to **MainActivity.java**:

MainActivity.java

```
package com.oreillyschool.android1.threads;

import android.app.Activity;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.os.AsyncTask;
import android.os.Bundle;
import android.view.View;
import android.widget.ImageView;
import android.widget.ProgressBar;

public class MainActivity extends Activity {

    private ImageView image;
    private ProgressBar progress;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        progress = (ProgressBar) findViewById(R.id.progress);
        image = (ImageView) findViewById(R.id.image);
    }

    public void onLoadImageClicked(View view) {
        // show the progress and hide the image
        image.setVisibility(View.GONE);
        progress.setVisibility(View.VISIBLE);

        image.setImageBitmap(downloadImage());
        // show the image and hide the progress
        image.setVisibility(View.VISIBLE);
        progress.setVisibility(View.GONE);

        new AsyncTask<Void, Void, Bitmap>() {
            @Override
            protected Bitmap doInBackground(Void... params) {
                return downloadImage();
            }

            @Override
            protected void onPostExecute(Bitmap bitmap) {
                image.setImageBitmap(bitmap);

                // show the image and hide the progress
                image.setVisibility(View.VISIBLE);
                progress.setVisibility(View.GONE);
            }
        }.execute();
    }

    private Bitmap downloadImage() {
        final long start = System.currentTimeMillis();

        // wait 5 seconds (5000 milliseconds) until proceeding
        while (System.currentTimeMillis() - start < 5000) {
        }

        return BitmapFactory.decodeResource(getResources(), R.drawable.ic_launcher);
    }
}
```

AsyncTask is an abstract class, meaning we have to define its implementation. Typically, you would create a new class that extends AsyncTask, but that's not always necessary. You can define an in-line class implementation (also known as an anonymous class). This is especially useful when your implementation is going to be short and relatively uncomplicated:

OBSERVE:

```
new AsyncTask<Void, Void, Bitmap>() {
    @Override
    protected Bitmap doInBackground(Void... params) {
        return loadImage();
    }

    @Override
    protected void onPostExecute(Bitmap bitmap) {
        image.setImageBitmap(bitmap);

        // show the image and hide the progress
        image.setVisibility(View.VISIBLE);
        progress.setVisibility(View.GONE);
    }
}.execute();
```

AsyncTask has three generic class parameters that must be defined in any implementation. The first parameter (**Params**) is used to define any parameter input. In our example, we didn't need any parameters so we defined this parameter as **Void** (Null). This generic class parameter corresponds to the class type of objects passed to the **execute()** method and received in the **doInBackground()** method. The ellipses (...) at the end of the type in the **doInBackground()** method is called "varargs" (variable-length argument). This syntax allows any number of arguments of the class type to be sent to the method, which will then be combined into an array in the method automatically. Had our image-loading method been implemented to download an image from the internet correctly, then we probably would've defined this generic as the **String** or the **URI** class. In that case, our implementation might have looked something like this:

OBSERVE:

```
new AsyncTask<String, Void, Bitmap[]>() {
    @Override
    protected Bitmap doInBackground(String... params) {
        Bitmap[] bitmaps = new Bitmap[params.length];
        for (int i=0; i<params.length; i++) {
            bitmaps[i] = loadImage();
        }
        return bitmaps;
    }

    @Override
    protected void onPostExecute(Bitmap[] bitmaps) {
        loadBitmapsIntoImageViews(bitmaps);
    }
}.execute(url1, url2, url3);
```

The second generic parameter for AsyncTask is used to track progress. We usually use a numeric primitive for this generic, like an **Integer** or a **Float**, but you can use whatever class you like, including a String or even your own custom class. (We'll discuss progress tracking further a bit later in the lesson.)

The third parameter is used to define the class type of the result of the work done in the **doInBackground()** method. This parameter is used as the return type of **doInBackground()**, as well as the parameter type for the **onPostExecute()** method. All of the generic parameters are technically optional. If you'd rather not use any in your application, you can just define each of them as **Null** and ignore them in your code.

doInBackground() is the only method in AsyncTask that we must define in our implementation (that is, the only abstract method). So, you might be wondering why we return the final result value in **doInBackground()** and then handle assigning the image to the ImageView in **onPostExecute()**. We do that because in Android you cannot interact with View components that are attached to the view hierarchy from any thread other than the main "UI" thread. The **doInBackground()** method runs on a new thread that was created specifically for the AsyncTask class and so it cannot assign the bitmap to the ImageView component. However, the **onPostExecute()** method is guaranteed to run on the UI thread, so we can assign the data to the ImageView in

that method safely.

Tracking Progress

When an application is performing extensive work, it will often report progress to the user in the form of a progress bar. The second generic parameter in `AsyncTask` helps with the implementation of progress tracking in a thread-safe manner. Again, the `doInBackground()` method does not run on the UI thread, so it cannot be used to update any views. Fortunately, there is another helper method that runs on the UI thread that we can override to handle progress updates. Make these changes to **MainActivity.java**:

MainActivity.java

```
package com.oreillyschool.android1.threads;

import android.app.Activity;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.os.AsyncTask;
import android.os.Bundle;
import android.view.View;
import android.widget.ImageView;
import android.widget.ProgressBar;

public class MainActivity extends Activity {

    private ImageView image;
    private ProgressBar progress;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        progress = (ProgressBar) findViewById(R.id.progress);
        image = (ImageView) findViewById(R.id.image);
    }

    public void onLoadImageClicked(View view) {
        // show the progress and hide the image
        image.setVisibility(View.GONE);
        progress.setVisibility(View.VISIBLE);

new AsyncTask<Void, Void, Bitmap>() {
        new AsyncTask<Void, Integer, Bitmap>() {
            @Override
            protected Bitmap doInBackground(Void... params) {

                private Bitmap downloadImage() {
                    final long start = System.currentTimeMillis();

                    // wait 5 seconds (5000 milliseconds) until proceeding
                    int progress = 0;
                    int current = 0;
                    publishProgress(progress);
                    while ((current = (int)(System.currentTimeMillis() - start)) < 5
000) {
                        current = (int) ((float)current * 100 / 5000);
                        if (current > progress) {
                            progress = current;
                            publishProgress(current);
                        }
                    }

                    return BitmapFactory.decodeResource(getResources(), R.drawable.i
c_launcher);
                }

            @Override
            protected void onProgressUpdate(Integer... values) {
                progress.setProgress(values[0]);
            }

            @Override
            protected void onPostExecute(Bitmap bitmap) {
                image.setImageBitmap(bitmap);
            }
        }
    }
}
```

```

        // show the image and hide the progress
        image.setVisibility(View.VISIBLE);
        progress.setVisibility(View.GONE);
    }
}.execute();
}

private Bitmap downloadImage(){
    final long start = System.currentTimeMillis();
    // wait 5 seconds (5000 milliseconds) until proceeding
    while (System.currentTimeMillis() - start < 5000) {
    }
    return BitmapFactory.decodeResource(getResources(), R.drawable.ic_launcher);
}
}

```

Here, we moved the `downloadImage()` method we defined earlier into the anonymous class implementation of `AsyncTask` and made a couple of changes to it. Now we'll make one minor change to the `ProgressBar` in the XML layout in order to display the progress. Modify the `ProgressBar` in **activity_main.xml** as shown:

/res/layout/activity_main.xml

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context=".MainActivity" >

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="onLoadImageClicked"
        android:text="Load Image" />

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center" >

        <ProgressBar
            android:id="@+id/progress"
            style="?android:attr/progressBarStyleHorizontal"
            android:minWidth="200dp"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:indeterminate="false" />

        <ImageView
            android:id="@+id/image"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:scaleType="center"
            android:visibility="gone" />

    </RelativeLayout>

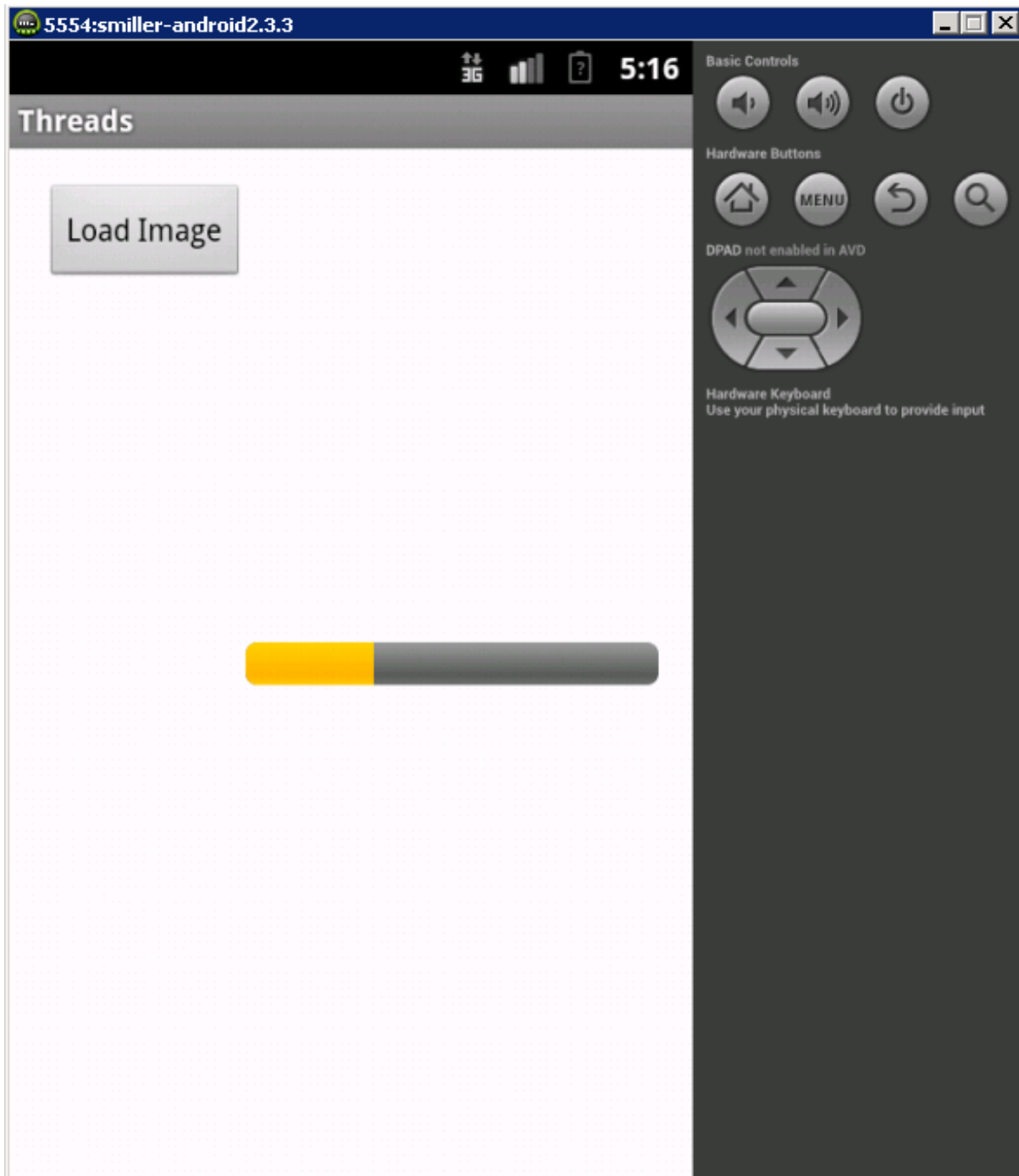
</LinearLayout>

```

Note

You might be wondering about the curious syntax we just used for the style attribute of the `ProgressBar`. This is a unique way of referencing a dynamic style value. Don't worry too much about it for now, we'll hit this subject again later. Just remember that if you need to show a horizontal progress bar, use the style value `?android:attr/progressBarStyleHorizontal`.

Make sure all your changes are saved, and run the project in the emulator. Now when you load the image you should see the progress bar crawl its way across the screen during the five seconds the download method takes to finish:



We moved the download logic inside of the anonymous class so that we could make calls to the `publishProgress()` method. The Integer value we passed to `publishProgress()` gets sent to the `onProgressUpdate()` method that we have implemented as well now. `onProgressUpdate()` runs on the UI thread, so we can update the progress of the `ProgressBar` view component here safely. The `Integer` generic value is also defined with `varargs`, so it is put into an array automatically. We only send one value at a time to `publishProgress()`, so we can grab the first item out of the array (`values[0]`) safely each time.

We've also added some code to our download method to make sure we don't call `publishProgress()` on

every single iteration of the **while** loop. Calling **publishProgress()** each time is unnecessary (since the progress won't always increment enough to even be noticeable on the progress bar), and could cause the application to slow down if we overwhelm the UI thread with progress updates. That would defeat our purpose and make the application look broken to users, so we've added a small check to make sure the progress has increased by at least a factor of 1%.

Another method that can be useful when you're implementing an AsyncTask class is the `onPreExecute()` method. It's guaranteed to run on the UI thread just like `publishProgress()` and `onPostExecute()`. This method is a great place to implement any setup for a progress bar or show a notification to the user that a background action is about to begin.

Wrapping Up

This lesson was relatively brief, but it may be the most important lesson yet. If you know how to write an Android application that implements threading properly, it can make the difference between getting featured on the Android Market and getting a slew of lowly one-star reviews. I'm confident that you can work with the AsyncTask and avoid the dreaded ANR dialog now. Good work!

See you in the next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Styles and Themes

Welcome back! In this lesson we'll go over the various ways to style Android views and components. Styling is a significant portion of application development and is way too dense to be covered in just one lesson. I'm going to focus on the basics of styling that will apply to most components. By the end of the lesson, you'll feel comfortable modifying the default Android style for applications.

Introduction to Styling

Like most elements in Android, there are a lot of different ways to go about implementing styles for your Android components. We'll cover a few of the more common ones. Create a new project named **Styling**, with the package name **com.oreillyschool.android1.styling**, and assign the project to the **Android1_Lessons** working set.

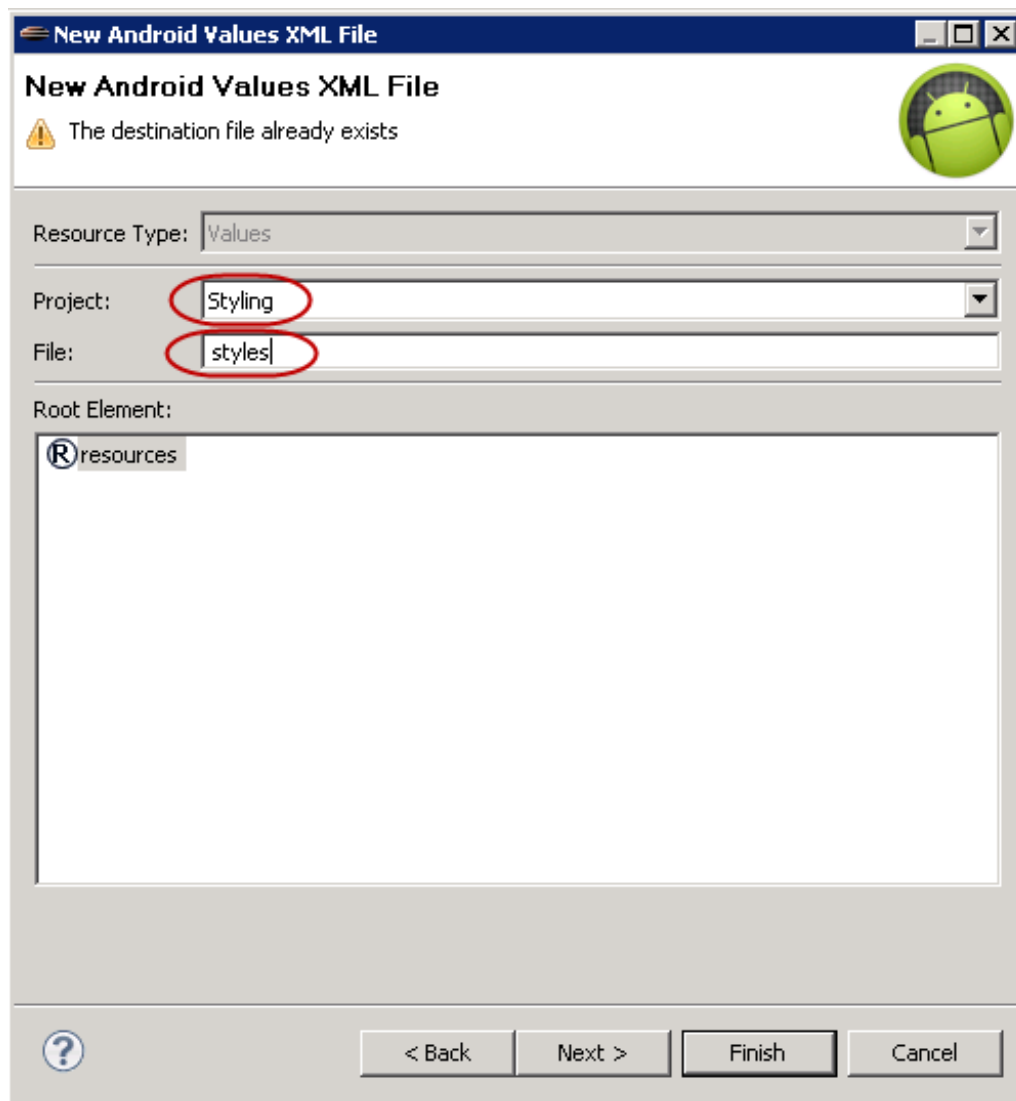
Defining Styles

Perhaps the easiest way to define styles on a component is directly on the view component XML definition. We've already done a little bit of this in previous lessons, and you might have noticed the options yourself if you used code assist in Eclipse to write your XML. Changing a style directly on a component is an efficient way to update a single component, but when you want to style an entire application, that can get tedious. To manage the style of an entire application, you'll want to use the styles and themes resources.

Note

We use a particular convention to name all of our XML files in the **/res/values** folder. You can use whatever file names you like; just make sure the root XML tag is **<resources>**.

Let's add a styles XML resource file to our project and define some initial values. Select **File | New | Other | Android XML Values File** to create a file named **styles** as shown:



Answer **Yes** when prompted to overwrite the existing file. When you finish, open the **styles.xml** file in the **/res/values** folder, switch to the manual edit (styles.xml) sub-tab and make these changes:

```
/res/values/styles.xml

<?xml version="1.0" encoding="utf-8"?>
<resources>

    <style name="MyButtonStyle">
        <item name="android:background">#aa0000</item>
        <item name="android:textColor">#000000</item>
        <item name="android:drawableLeft">@drawable/ic_launcher</item>
    </style>

</resources>
```

This is a basic style definition. Every style definition needs just one attribute, a **name**, which is used to reference the style using the **@style** syntax. All children of the **style** tag must be **item** tags, with **name** attributes of their own. The **item** tag's **name** attribute must reference a style property of the View component that this style will modify. There is no component type-checking handled here, so be careful that the styles you define are actually applicable to the component you are styling, otherwise you might be unpleasantly surprised when you change styles later and nothing changes in the view!

The **android:background** style is common to all view components in Android. It can accept both drawable resources and colors. In fact, just about any style that accepts a drawable can accept a color definition instead. However, the inverse is not true. For example, the **android:textColor** attribute must be a color definition and not a drawable reference.

Next, we'll use this style in a layout view. Open **activity_main.xml** and make these changes:

```
/res/layout/activity_main.xml

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context=".MainActivity" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />

    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="My Styled Button"
        style="@style/MyButtonStyle" />

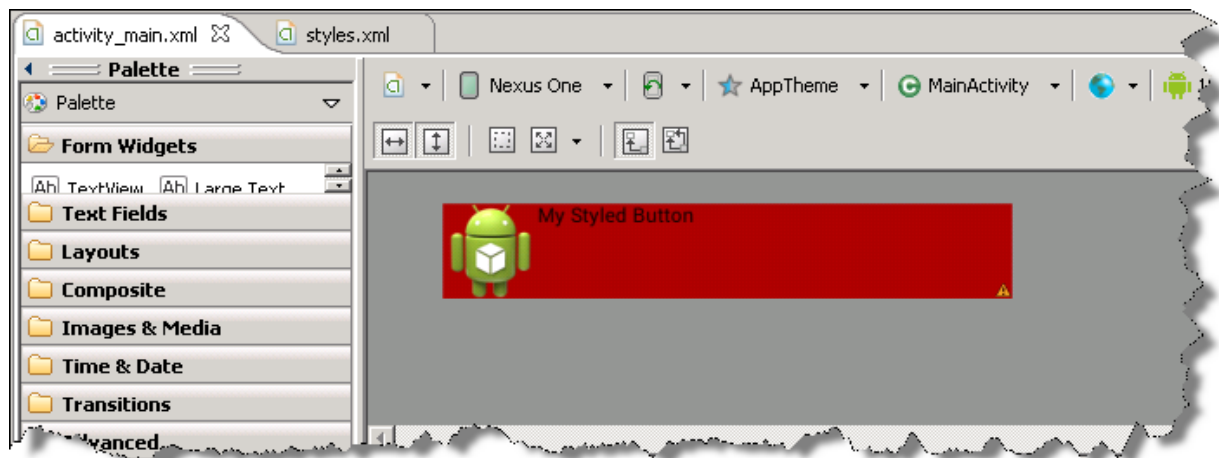
</RelativeLayout>
```

Note

You should be able to test the styles of most code you change in this lesson just by looking at the "Graphical Layout" sub-tab of activity_main.xml in the XML editor. Sometimes the "Graphical Layout" is unable to generate the view properly and you'll need to run the application in the emulator, but for most minor changes you make, you won't need to wait for the emulator to test them.

The style attribute is a unique attribute for view components in XML layouts because it doesn't use the "android" namespace (notice it's called "style" instead of "android:style"). While most layout XML attributes correspond to a property on the respective view class, style is not a property of any view. This is important; it means that in order to change the styles of a view component at runtime you will have to change each individual style. There's no way to update the XML-defined style property at runtime. Also, note that we use the **@style/<style name>** syntax to reference the style we defined in styles.xml.

Using the "Graphical Layout" mode of the editor, or running the application in the emulator, we can now test to make sure that our styles defined in **styles.xml** do indeed get assigned to the button in the view.



Defining Themes

Using the style attribute on a XML view makes it convenient for reusing styles that you would potentially apply to multiple views. But what if you just want to apply the style to every component in your application generically? This is where **themes** come into play.

Themes are defined exactly like styles—they even use the same XML node name of **<style>**. The difference is in how you use them. We'll get to that, but first let's create a new file to contain our themes for the application. This is a standard convention used to organize the definitions and make it easier to find each resource later; if you *really* wanted to, you could define all your styles and themes in the same file.

Okay let's get to work. Remove the style tag from Button in **activity_main.xml**:

```
/res/layout/activity_main.xml

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context=".MainActivity" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />

    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="My Styled Button"
        style="@style/MyButtonStyle" />
    />
</LinearLayout>
```

In the "Graphical Layout," verify that the style has been removed and our button is back to looking like a regular button. Next, create another new Android Values XML file named **themes.xml**, and then make these changes:

```
/res/values/theme.xml

<?xml version="1.0" encoding="utf-8"?>
<resources>

    <style name="MyTheme" parent="@android:style/Theme">
        <item name="android:buttonStyle">@style/MyButtonStyle</item>
    </style>

</resources>
```

Now let's use the theme. As I mentioned before, the difference between themes and styles isn't in the way you define them, but how you use them. To use a theme, you define it for either an activity or the entire application. This is all done in the **AndroidManifest.xml** file.

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.oreillyschool.android1.styling"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="10"
        android:targetSdkVersion="10" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/MyTheme" >
        <activity
            android:name="com.oreillyschool.android1.styling.MainActivity"
            android:label="@string/app_name" >
            <intent-filter >
                <action android:name="android.intent.action.MAIN" />

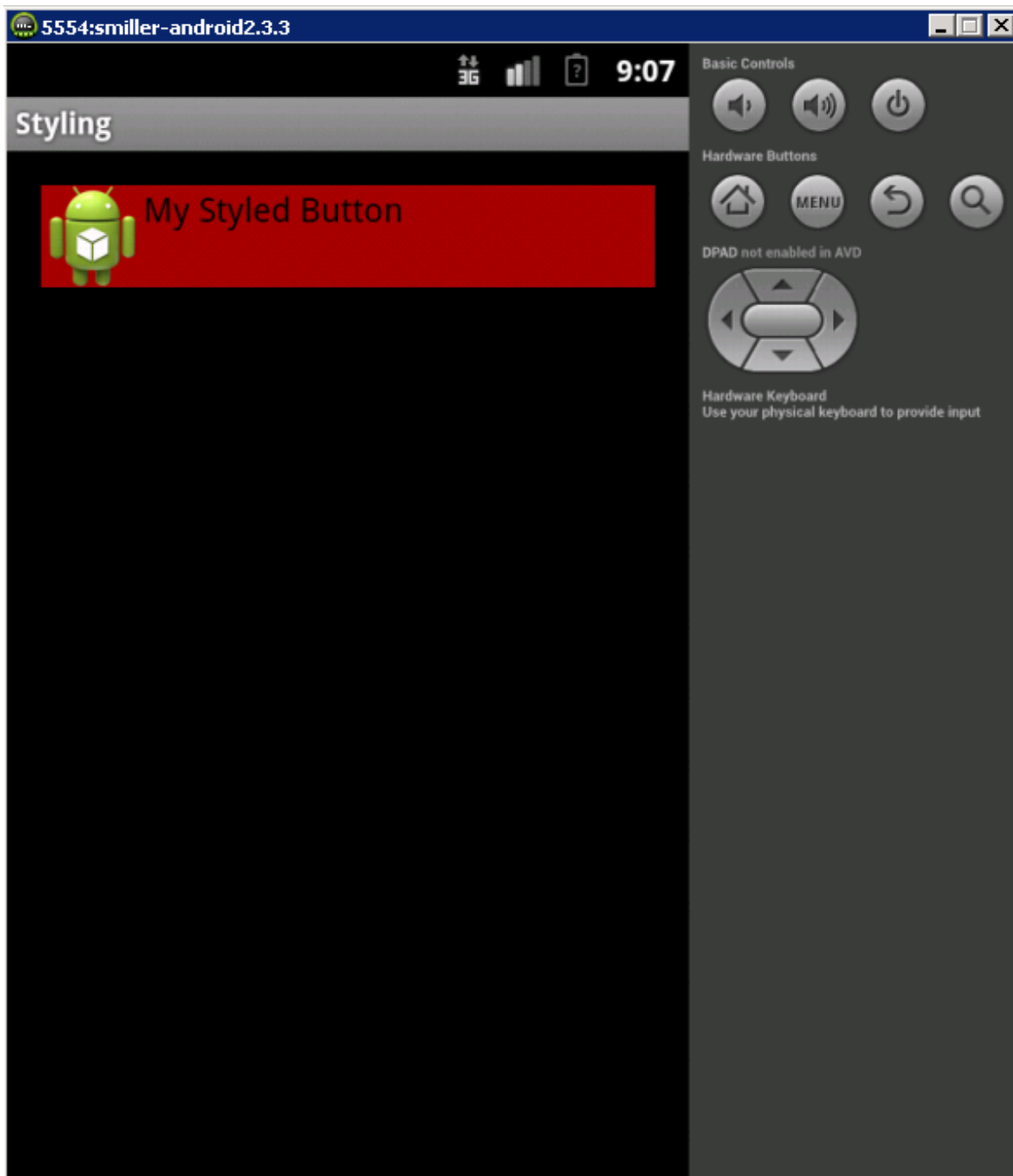
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Here, we defined our theme on the **<application>** tag. This will ensure that every activity we create in this application will have its core styles defined by the theme resource **MyTheme**. (If you wanted to have a specific alternate theme for just one Activity, you could also add an **android:theme** attribute to the **<activity>** tag. Then any styles defined in the application theme would be overridden by the styles from the activity's theme.)

The "Graphical Layout" viewer for XML layouts occasionally has a hard time loading themes correctly. You'll need to use the theme dropdown located in the top right to select your new theme **MyTheme**:

□

If **MyTheme** doesn't show up initially, try changing something else (like the Android SDK version above the theme dropdown) to get it to refresh, or close and reopen the editor. If all else fails, you can just run the application to test it on the emulator:



Let's look at the theme we defined in **themes.xml** in more detail:

OBSERVE:

```
<resources>

    <style name="MyTheme" parent="@android:style/Theme">
        <item name="android:buttonStyle">@style/MyButtonStyle</item>
    </style>

</resources>
```

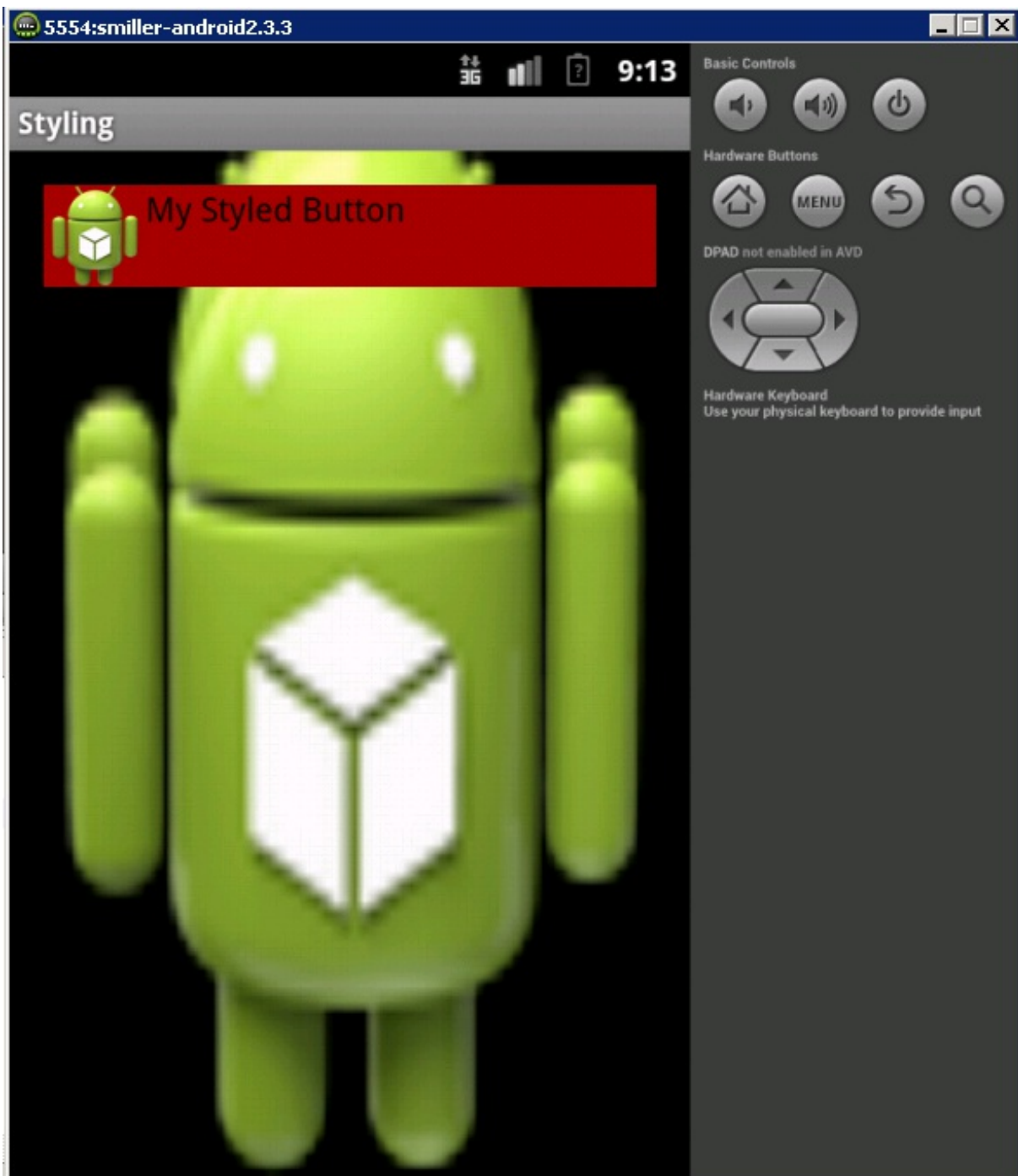
The only item we've added to this theme/style is **android:buttonStyle**, which references the style we defined in styles.xml named **MyButtonStyle**. This item works only in a style that is used as a theme. **android:buttonStyle** defines the style that is used on buttons. There's a list of all items you can define in a theme (and descriptions of what they will do) available on the [Android developer documentation site](#).

Here are some of the more common items (with the acceptable value types in parentheses) that you might

consider overriding when designing themes for your own applications:

- android:windowBackground (drawable or color)
- android:windowNoTitle (boolean)
- android:buttonStyle (style)
- android:tabWidgetStyle (style)
- android:checkboxStyle (style)
- android:listViewStyle (style)
- android:listDivider (style)
- android:listPreferredItemHeight (dimension)
- android:dialogTheme (style/theme)
- android:textAppearance (style)
- android:textAppearanceButton (style)
- android:textColorPrimary (drawable or color)
- android:textColorPrimaryInverse (drawable or color)
- android:textColorSecondary (drawable or color)
- android:textColorSecondaryInverse (drawable or color)

Another difference between styles and themes is that a theme will cascade, but a style will not. If you're familiar with CSS files in web development, you're familiar with the concept of cascading styles. In Android, if you define a style for a **LinearLayout**, such as a background image or color, that style will not cascade to its children. That means you don't have to worry that all the sub-views will get assigned the same background or color. However, if you define a style item such as **android:background** in a theme, and then use that theme on an Activity, every view in that activity's view hierarchy will inherit that same background (if they don't manually override with a different background). Be careful about what you define in themes. As a general rule, you should never define **android:background** in a theme. Go ahead and define **android:background** in our current application giving it the **@drawable/ic_launcher** as the value and run the application. Behold the disastrous results!:



When you want to define a global background that shows up as the background of every Activity of your application, but not every view component, use the **android:windowBackground** property instead. That way you can be certain that your selected background shows up only as the background to your activity window, and won't cascade to any view components in the view hierarchy.

Style Inheritance

The last concept I want to discuss regarding themes.xml is inheritance:

/res/values/themes.xml

```
<resources>

  <style name="MyTheme" parent="@android:style/Theme">
    <item name="android:buttonStyle">@style/MyButtonStyle</item>
  </style>

</resources>
```

Styles and themes can inherit items from other styles in two different ways. The first way is demonstrated in our theme with the **parent** attribute. Using the **parent** attribute on a style, we can inherit all of the elements of a style defined in the Android SDK package. This is recommended, especially for themes, so that you receive all the default styles that you are used to seeing, and then you can select which individual items to override. The following are some of the more popular Android SDK themes you can use to parent your own themes:

- @android:style/Theme
- @android:style/Theme.Black
- @android:style/Theme.Black.NoTitleBar
- @android:style/Theme.Black.NoTitleBar.Fullscreen
- @android:style/Theme.Light
- @android:style/Theme.Light.NoTitleBar
- @android:style/Theme.Light.NoTitleBar.Fullscreen

The second method of inheriting from another style is demonstrated in the list above. You can prefix the name of your style with another style name and a period. You can use this method to create alternates of a style. For example, to create a sub-style of our earlier **MyButtonStyle** you could name it **MyButtonStyle.Large**, and then have another style inheriting from that one named **MyButtonStyle.Large.Red**, and so on.

Note Prefix inheritance only works for other styles that you have defined in your application. In order to inherit from Android SDK styles, you must use the **parent** attribute.

Direct Theme References

While themes will style the default components with the styleable items available to the Theme class automatically, on occasion you might want to pull a value directly from a theme to be assigned to a different View. We used this syntax in another lesson when we defined the style of the **ProgressBar** component. The syntax for referencing a style item from a theme is written either, "**?attr/themeAttribute**", or with the Android SDK namespace, "**?android:attr/themeAttribute**".

Let's practice using our button code. First, remove the theme reference that overrides the default button style (and the one we added to create the ugly background) in **themes.xml**:

```
/res/values/themes.xml

<?xml version="1.0" encoding="utf-8"?>
<resources>

    <style name="MyTheme" parent="@android:style/Theme">
        <item name="android:buttonStyle">@style/MyButtonStyle</item>
        <item name="android:background">@drawable/ic_launcher</item>
    </style>

</resources>
```

Next, make the following changes to **activity_main.xml**:

/res/layout/activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context=".MainActivity" >

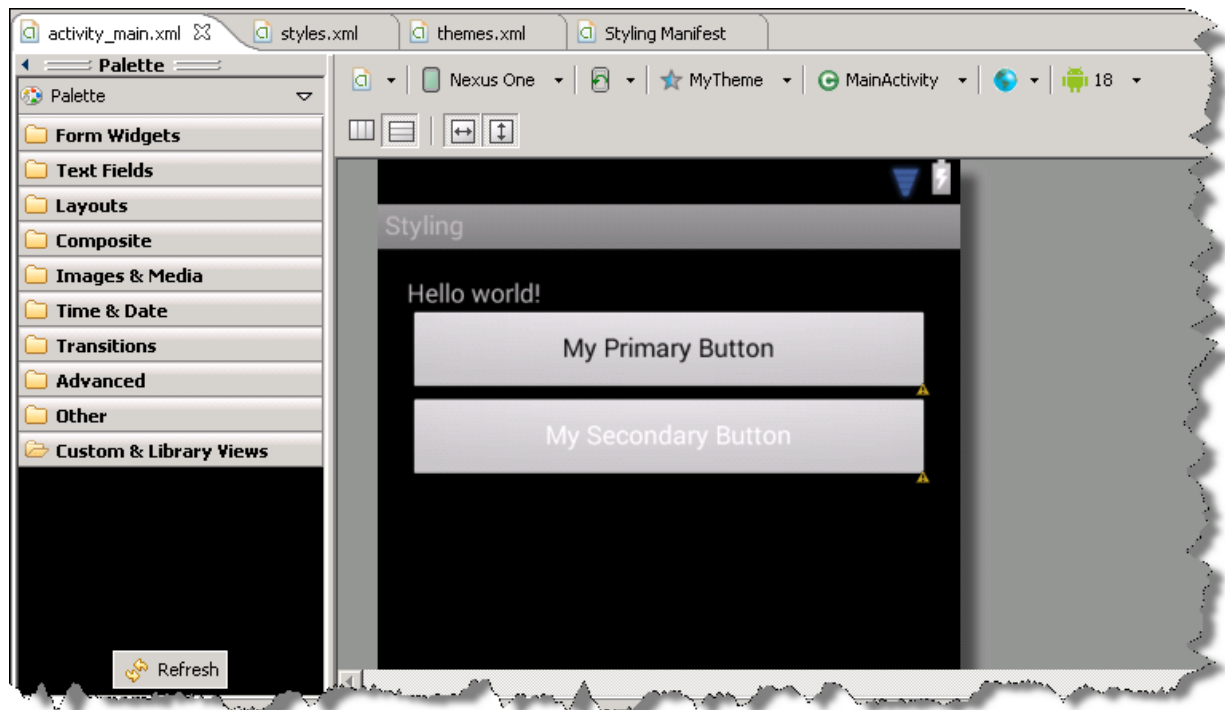
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />

    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="My StyledPrimary Button" />

    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textColor="?android:attr/textColorPrimary"
        android:text="My Secondary Button" />

</LinearLayout>
```

Save and run it in the emulator or use the "Graphical Layout" view (though occasionally the "Graphical Layout" view will struggle when using these resources). Your view will look something like this:



Here we made a second button and told it to load the **textColorPrimary** theme value for its text color. The **textColor** on the Button component uses a dark black or near black color by default; we can see that in the first button. When we define the style as **?android:attr/textColorPrimary**, the color gets loaded as whatever is assigned to the **textColorPrimary** item in the current theme. We can even override **textColorPrimary** in our theme. Change **themes.xml** as shown:

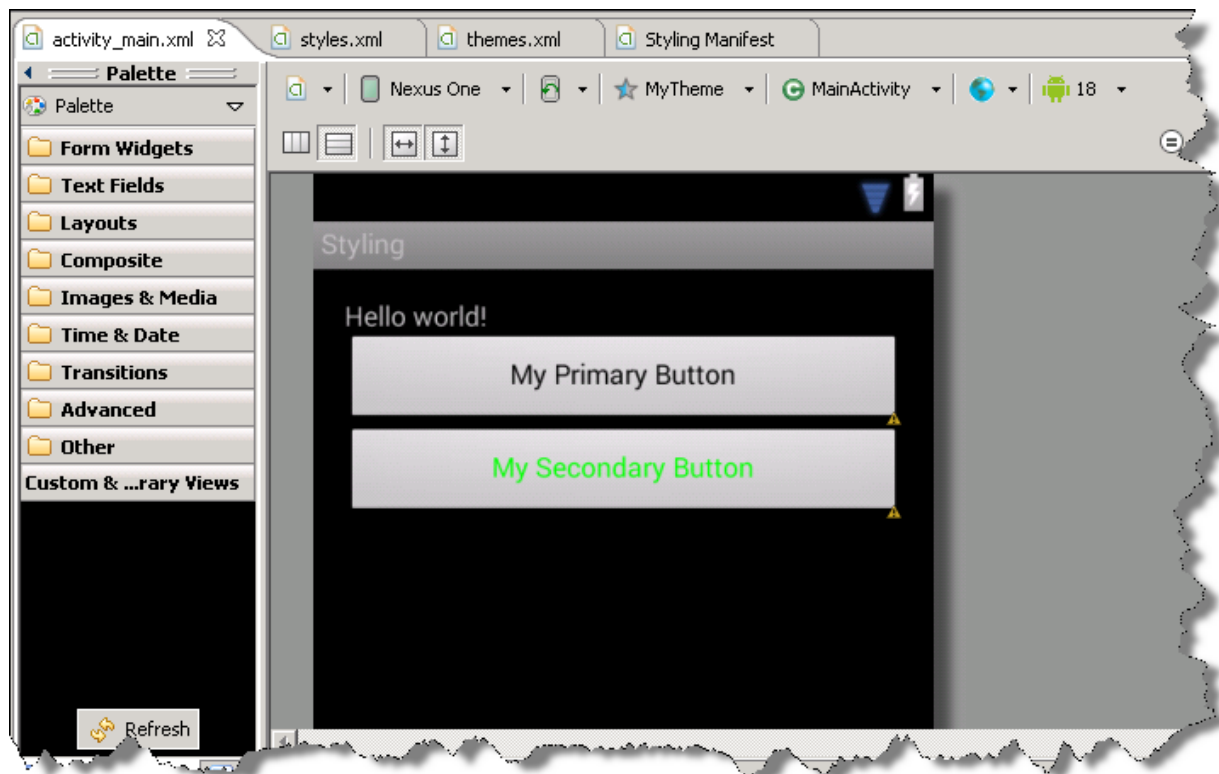
/res/values/themes.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <style name="MyTheme" parent="@android:style/Theme">
        <item name="android:textColorPrimary">#00ff00</item>
    </style>

</resources>
```

Here we gave the **textColorPrimary** item a hexadecimal color value of "#00ff00", which is a hideous bright green color. Now when we test the **activity_main.xml** view in the "Graphical Layout" or in the emulator, the second button's text is that color:



While overriding properties in this manner can be convenient for changing a style property in your application globally, do not rely on this technique. Changing a system property can have dramatic effects on your design and unintended consequences in weird places!

Learning to Learn

The most important tool you can have when it comes to styling may be the ability to look up what can and cannot be styled. The ADT plugin for Eclipse has come a long way and includes code hints for most XML properties. This may help you discover new attributes, but it can only get you so far. The best resource for learning about properties that can be used in styles for a View is on the class reference page for the view on the Android developer documentation site. For example, here is [the XML attributes section for the TextView component](#).

Also, check the parent components of a view to learn about other attributes available to components. Take a look at the (reference page for the Button component), for example. Even though there are many specific XML attributes available to Button, the reference page doesn't show any. That's because it doesn't have any specific unique styles available. All of Button's styles are inherited from its parent components (TextView and its parent, View).

Another more concise list can be found on the ([R.styleable resource page](#)). This page actually lists the properties available to every standard component in the Android SDK, and is definitely worth putting in your bookmarks.

Wrapping Up

Having a good design for your Android application is crucial for its success on the Android market. As the platform has

matured, users have come to expect a high-quality look and feel in their applications. The Android styles and design standards seem to change with each new release of the Android SDK, but the core methods to styling remain the same. The skills you have now will help you keep current with the latest styling techniques through each update to the Android SDK.

You're almost done! Great work so far. In the next lesson, you'll be completing your final project for the course. You'll have a chance to show your stuff there—I'm looking forward to seeing what you can do!

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Android Final Project

Final Project

Congratulations on completing the lessons! For your final project you will create your very own Android application (surprised?). The type of application you create is entirely up to you, but just in case, here are some ideas you can use for your project:

- A basic note-taking application—supports creating multiple notes, editing notes, and deleting notes.
- A "To Do" list application—slightly more involved than the note taking application, the "To Do" application supports creating, editing, and deleting multiple "To Do" lists. Each list can be modified to add, edit, or remove "To Do" items. The items can also be "checked" off when completed and the time the items were completed is recorded (and displayed in the view).
- Hangman!—the Hangman application will implement the game Hangman using the common view components (no need for intense graphics here). The word used in the game is randomly chosen from a **string-array** XML resource. Users guess letters to spell out the word, and lose points for each incorrect guess. Use a simple points system, counting down from the appropriate number (usually 6, for a head, torso, two arms, and two legs), to keep track of remaining guesses and/or get creative with the view components or your own graphics which ultimately lead to the drawing of a hanged stick figure.
- Any type of internet data presentation application using a freely available public API (such as imgur, reddit, or yahoo weather). There are many other publicly available APIs (such as Flickr, all Google APIs, and Twitter), but they usually require you to sign up for an API key (feel free to do that if you like). This type of Application will need to implement a data interpreter such as an XML or JSON parser. If you are unfamiliar with using libraries for these interpreters, then you might not want to tackle this type of application right off the bat. If you do choose this type of application, make sure you adhere to the policies for the API and give proper attribution to the source of the data.

Whichever you choose, your application must meet these requirements:

- Functions on Android devices.
- Implements at least three Activities, each with a unique view layout.
- At least two Activities share data between each other using the proper Intent passing methods.
- Implements at least one ListView, with its own custom adapter and custom view layout for the list items.
- Implements at least one Dialog using the **new** process with the support library.
- Implements a SharedPreferences object (implementing a PreferenceActivity is optional, but can count towards one of your three Activities).
- Implements a SQLite database for caching data between application sessions.
- All SQLiteDatabase usage (such as query, insert, and delete) should be used inside of an AsyncTask.
- All internet usage (if implemented) is used inside of an AsyncTask.
- All hard-coded strings are loaded from string XML resources.
- Use themes and styles via XML resources for all appropriate styling properly.

Make an application that you are proud to have created! Keep your code clean, organized, and bug-free! You might even consider publishing your work on the Android Market when you are finished. Thanks for taking the course and good luck!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.