Android: Get started

Writing a simple Android program is a quick project. Set up, produce and test a countdown timer app in this first instalment of our new series.

Our expert

Juliet Kemp is now on her second Android phone and has difficulty recollecting how she coped without one. Sadly it doesn't quite write the code *for* you yet, although it's probably only a matter of time.

As Android smartphones proliferate (and the chance that you have one of your own increases), writing a short program for them becomes a fun project, even if your coding experience is minimal. Unlike other smartphone operating systems, Android is open source and runs on a Linux kernel, with a collection of Android-specific Java libraries acting as intermediaries between the kernel and the software developer. All the development tools are available online, and if you're familiar with even basic Java, then the comprehensive API and documentation make it easy to get started.

We'll assume a few basic familiarities in this tutorial:

1 Some notion of Java (we won't explain the basics of how classes and methods work), although even if you have no Java experience you've got a fair chance of being able to follow along anyway.

2 Some basic programming concepts (compiling code, looking for bugs and so on).

3 A very basic grasp of Extensible Markup Language (XML).

However, even if your knowledge of the above is a little shaky, one of the best ways to start finding out about it is to get stuck into some code – so keep reading!

"The best way to find out about it is to get stuck into some code."

You can either write your Android code in *Eclipse* (a Java development environment) with the Android plugin or take the slightly more old-school route and write and compile everything by hand. *Eclipse* does take a bit of the work out of Java boilerplate construction, but especially when you're starting out it's not that hard to go for the do-it-yourself option, and it does give you a much clearer idea of what's going on under the hood. We'll use the DIY route here, partly for that extra control, partly because setting up *Eclipse* is easily a tutorial all of its own, and perhaps partly because we're a bit inclined to the old-school ourselves.

Getting set up

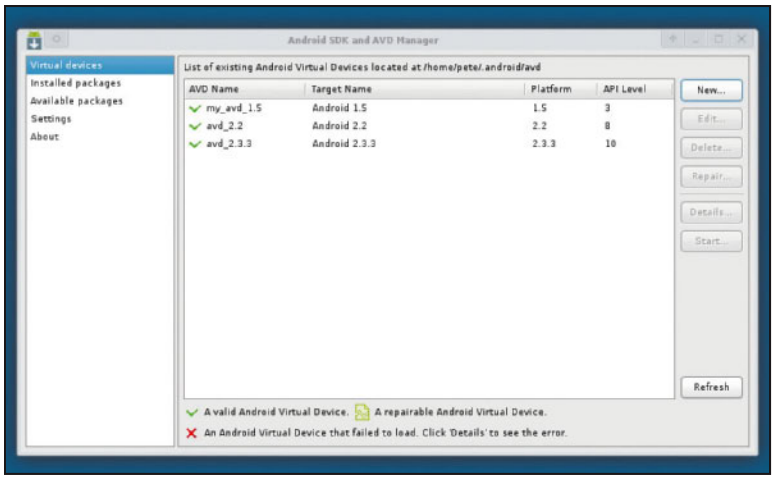
The first step in creating your app is to download the Android Software Development Kit (SDK) starter kit from the Android developer site, **http://developer.android.com/index.html**. At the time of writing, the current version is rev 10. Download and unzip it, then put the directory wherever you fancy keeping it. Check that the permissions are set up correctly, so that you can access the directories and tools, and then have a look at the directory structure. You can get at the offline documents from the **docs/offline.html** page. You'll need version 1.5 or 1.6 of Java to work with the most recent version of Android: 1.6 is available as the **openjdk-6-jdk** package for Debian or Ubuntu. Note that Android won't run with **gcj** (the GNU Compiler for Java). You also need to install Apache Ant version 1.8 (the **ant** package on Debian stable and Ubuntu 10.10; **ant1.8** on Ubuntu 10.04) to compile Android binaries.

It's a good idea to add **tools/** and **platform-tools/** to your **$PATH**, so you can get at these tools directly. To do so, edit your **~/.bashrc** to include this line, substituting your SDK directory for **<sdk>**:

```
export PATH=${PATH}:<sdk>/tools:<sdk>/platform-tools
```

Then type **source ~/.bashrc**. That gets the basics set up, but the SDK starter kit includes only the core tools, not any of the Android platforms. You can use the **android** graphical SDK and the Android Virtual Device (AVD) manager (which can be found in **tools/**) to grab whichever platforms and other packages you want.

Take a look at the Available Packages tab. You'll need to install at least one platform (see the Android Platforms box for further information) and in this tutorial we'll be developing against Android 2.2 (API 8). You should make sure that you have the Documentation, Tools and Platform-tools packages ticked, and you can also grab the Samples. Later on, you might want to install the Google APIs add-on, which gives you Maps access. Hit Install Selected and go and make a cup of tea while it all downloads. After that, you'll be ready to start developing your application.



List of existing Android Virtual Devices located at /home/pete/.android/avd

AVD Name	Target Name	Platform	API Level
✓ my_avd_1.5	Android 1.5	1.5	3
✓ avd_2.2	Android 2.2	2.2	8
✓ avd_2.3.3	Android 2.3.3	2.3.3	10

✓ A valid Android Virtual Device. A repairable Android Virtual Device.
✗ An Android Virtual Device that failed to load. Click 'Details' to see the error.

› **The Android SDK and AVD manager, which is showing virtual devices.**

The **android** tool used on the command line will create an empty project for you, with the correct structure for it to compile into a working Android binary. Create a working directory for your Android code projects, and **cd** into it, then generate the project:

```
mkdir ~/android_code/
cd ~/android_code/
android create project --target android-8 --name countdown
--path ~/android_code/countdown --activity
CountdownActivity --package com.example.countdown
```

So, what do all those options mean?

1 **target** Identifies the platform that you're building against. Use **android list targets** to see what's available. The numerical ID will depend on what you have installed locally, but the name of the target will stay the same – **android-8** is Android 2.2, currently the most popular.

2 **name** Sets the project name.

3 **path** The project directory; this will be created if necessary.

4 **activity** This sets the name of your main class of your project. There's more to Activities than this, but because this project only has one Activity, there's no need to investigate further for now.

5 **package** This identifies your project namespace. Like Java packages, the rule is that you take your internet domain name and reverse it to generate the package name; thus **com.example.countdown** from the **example.com** domain. If you don't have a domain, you can use **local.myname.countdown**, but you risk namespace collision.

If you **cd** into the new directory, you can check out the directory and project structure. Source code lives in **src/**; **res/** contains various package resources, including layout. After that, **build.xml** is the build file, which you shouldn't need to edit by hand, and the other important file is **AndroidManifest.xml**, which sets up the application's components and structure, and declares the necessary libraries, the platform and other similar information. The online docs provide more information about this file.

At the moment, this is just an empty project, with no functional code, but it does still compile. Run:

```
ant debug
```

from the top-level project directory to generate a **bin/countdown-debug.apk** file. The **debug** build target is used when developing; we'll look at generating a 'real' program next time. Now we can start to write some code.

Your first Android coding

Open the **CountdownActivity.java** file, created for you under the **src/** directory, in your preferred text editor to get started.

First, at the start of the class you need to declare a few private variables (variables accessible only within this class, not to any other classes) that you'll use later on:

```
public class CountdownActivity extends Activity {
  private static final int MILLIS_PER_SECOND = 1000;

  private TextView      countdownDisplay;
  private CountDownTimer timer;
}
```

The first one is a constant declaring how many milliseconds there are in a second (that would be 1000). The second two are internal variables for use in later methods.

Every Android Activity has an **onCreate** method, the stub of which is created for you when the class is generated. This is called when the Activity is first created.

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

  countdownDisplay = (TextView) findViewById(R.id.
countdownDisplay);
  final EditText timeEntry = (EditText) findViewById(R.id.
timeentrybox);
  Button startButton = (Button) findViewById(R.id.
startbutton);
}
```

The initial **@Override** tells the compiler that this method overrides a method declaration in a superclass. The first line explicitly calls the **onCreate** method of the superclass, and then the second line grabs the view for this Activity. The view deals with the user interface (drawing things on-screen and managing any events), and we'll look at how Android manages layouts in the next section.

Display the details

The final three lines set up the display for the countdown (**countdownDisplay**, which was declared at the start of the class), the text entry field for the user to choose the countdown time (**timeEntry**), and the Start button

»

Android platforms

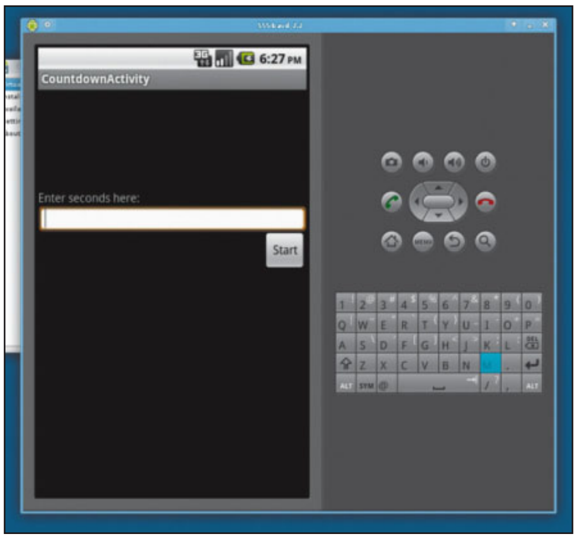
There are seven Android platforms (1.5, 1.6, 2.1, 2.2, 2.3, 2.3.3 and 3.0), each corresponding to a different API level, from 3 to 11. The page at **http://developer.android.com/resources/dashboard/platform-versions.html** has information on how many devices are running each platform. Most active devices are running 2.2 (this is probably what you have on your own phone); a fair few are still on 2.1; and there are a handful on 1.*. Android 2.3.3 is slowly being rolled out to newer phones and to the Google-controlled Nexus One, but it's unlikely to be released by other phone manufacturers for a while.

Any application developed for an earlier platform will be compatible with all newer platforms, because Android is entirely forwards-compatible. It's a good idea to use multiple virtual devices and test versions both backwards and forwards from your initial setup.

Be aware that Android 3.0 is different from the rest – it's been optimised specifically for tablets and other devices with larger screens. The UI is new and improved, and there's a new interaction model. Few devices are currently using 3.0, so unless you're one of those bleeding-edge users, you're better off sticking to developing on 2.*. However, if you want you can install and run the code in this tutorial on a 3.0 virtual device, and it will still work just fine due to the forwards-compatibility feature of Android.

Quick tip

Java requires a certain amount of boilerplate. Not all of the code is included in the code listings here, for reasons of space; check the **LXFDVD** for full code listings.



› **The text entry fields and buttons all laid out. The countdown display hasn't been kicked off yet, though.**

Abstract classes and interfaces

An abstract method is one that has a declaration, but no implementation, like this example:

```
abstract void onClick(View view);
```

A class that has abstract methods must be declared abstract itself (you can also declare a class to be abstract even if it *doesn't* have any abstract methods, if you want to).

Abstract classes can't be instantiated, so you can't just do:

```
OnClickListener pingListener = new
OnClickListener();
```

Instead, you have to create a subclass of it and instantiate that. This also means implementing all of its abstract methods or, alternatively, declaring the subclass itself also abstract (and thus subclassing *that* again to use it).

The subclass can be in its own class file, or in-line, as in the code here. **OnClickListener** has only a single method, **OnClick(View v)**, so that's all that must be implemented. In fact, **OnClickListener** is an interface, which is even more restricted than an abstract class; it can't contain *any* actual method bodies, only abstract methods and constants.

associated with that text entry field (**startButton**). Here, **timeEntry** needs to be **final** because it'll be referred to in an anonymous inner class in a moment.

As you can see in the code on the previous page, these all use the **findViewById** method – this sets up the screen layout, which will be covered in the next section, as will the **R.id** syntax used to refer to non-code resources.

Click the button

So far, the method has set up the layout and the necessary fields, but it's not actually doing anything with them. Next, let's set up a method that specifies what should happen when the Start button is clicked.

```
startButton.setOnClickListener(new View.OnClickListener() {

  public void onClick(View view) {
    String timeStringSeconds = timeEntry.getText().toString();
    int countdownMillis = Integer.parseInt(timeStringSeconds.
trim()) * MILLIS_PER_SECOND;
    showTimer(countdownMillis);
  }
});
```

The idea is that the user enters a time, and then clicks Start to set off the countdown. This method sets up an **OnClickListener** for the Start button; as the name might suggest, this 'listens' for the event of a click on the button. **OnClickListener()** is an interface class, so it must be subclassed. We could achieve this by creating a whole new class, **mySpecialButton**, outside the current class, but this would be overkill for a single button. Instead, we just do the whole thing in-line.

The user input box, **timeEntry**, has already been set up above; now we grab the text from it, turn it into a string and then turn that string into an integer and store it in **countdownMillis**. The **showTimer()** method (see below) is called with **countdownMillis** passed in as a parameter, and the button returns, so it's freed up to be clicked again.

This all works well if the user inputs an integer, but what happens if they try to input a string? This would, of course, be a bit dumb (what would it even mean to ask for a countdown from 'hello'?), but it's always best to expect users to try foolish (or simply inquisitive) things. We're not going to go far into the details of error-handling right now, but a basic catch-and-ignore looks like this:

```
try {
  int countdownMillis = Integer.parseInt(timeStringSeconds.
trim()) * MILLIS_PER_SECOND;
  showTimer(countdownMillis);
} catch (NumberFormatException e) {
  // method ignores invalid (non-integer) input and waits for
something it can use
  // A user popup here would be a good idea
}
```

With this in place, the code, as per the comments, simply ignores the invalid input. As a rule, it's not a good idea just to ignore an exception; if it's valid to do so (as it arguably is in this case, at least for now), then you should always include a comment to explain why you're doing it. A better option here, as the comment says, would be to generate a pop-up explaining the problem to the user, but that's beyond the scope of this tutorial.

Show the timer

The one thing now left to do with the code is to write the method that shows the countdown. Usefully, Android has a **CountDownTimer** class for us to use:

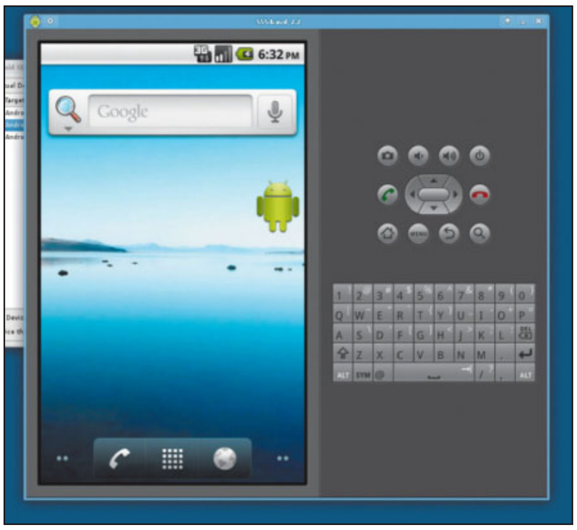
```
  private void showTimer(int countdownMillis) {
    timer = new CountDownTimer(countdownMillis, MILLIS_
PER_SECOND) {
          @Override
      public void onTick(long millisUntilFinished) {
        countdownDisplay.setText("counting down: " +
          millisUntilFinished / MILLIS_PER_SECOND);
      }

          @Override
      public void onFinish() {
        countdownDisplay.setText("KABOOM!");
      }
    }.start();
  }
```

CountDownTimer has two arguments: milliseconds from now to count down to (hence multiplying our seconds value by 1000 to get **countdownMillis**, in the **onClick** method above), and how often a 'tick' happens. Setting this to 1000, as the constant does here, gives one tick per second, so **onTick** runs once per second, and the display changes once per second. You can play around with this value a bit to see what happens as it changes.

Again, this is an abstract class, and the **onTick(long millisUntilFinished)** and **onFinish()** methods must be implemented when it's subclassed. On every tick, the timer will display the message **counting down:** and the number of

Quick tip

Tick Scale Display To Real Size to get a more manageable display; we found eight inches to be a good screen size on a laptop.



› The Android emulator: the pretend device will respond to mouse movements, and there's a keyboard to the right. Click the middle-bottom icon to get to the applications list.

seconds left until the end. Once finished, it'll show **KABOOM!**. The very last line calls the **start()** method on the new timer, to kick off the countdown.

What happens if the user (being inquisitive again) enters a new value, and clicks the start button again, while the first timer is still running? There's nothing to stop a new timer from being created, which will fight with the first one for control of the display. A single extra line right at the start of the method will avoid that problem:

```
if(timer != null) { timer.cancel(); }
```

That's it for the code itself; now for the onscreen layout.

Layout and buttons

Layout for your main Activity class (this code has only one Activity anyway) is handled in the **res/layout/main.xml** file. All the program resources are kept here: images, layouts, string data and anything else that's not code. Layout and string data is stored in XML files, and if you're familiar with HTML, XML syntax shouldn't be too hard to translate. Check the full code listing to see the whole thing, but the first few lines look like this:

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

This is a **RelativeLayout** (one of several available layout types), which means that each view is set out in relation to the other views. It's vertically oriented, and content will wrap as necessary.

A view is any single object, so the text entry box, the countdown display and the button are all views.

> **"The SDK includes an emulator, so you can test it on your dev box."**

Each of them will have a section in the XML:

```
  <TextView
    android:id="@+id/countdownDisplay"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="50px"
    android:text=""
    />
```

At the start of the previous section, the **onCreate** method had some lines like this:

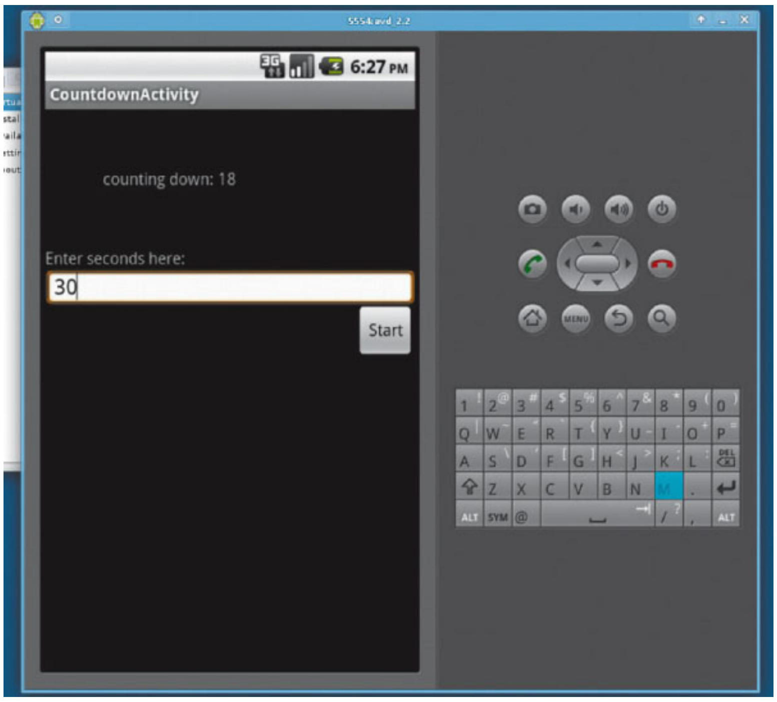
```
mCountdownDisplay = (TextView) findViewById(R.id.
countdownDisplay);
```

You locate a view by its **id** attribute, referred to as **R.id.idname**. You can also use the **id** attribute when referring to a view within the XML layout:

```
<TextView
  android:id="@+id/label"
      [ .... ]
  android:layout_below="@id/countdownDisplay"/>
```

The other attributes set up different aspects of the layout.

It's time now to compile everything with **ant debug** (from the **countdown** directory) and see if it runs. If you get any compile errors, check that you've imported all the necessary classes at the top of **CountdownActivity.java** (see the code listing on the **LXFDVD** for the full list of required classes). Once compiled, though, does it actually do what it's supposed to? The Android SDK includes an emulator, so you can test first on your dev box before installing to your phone.



› **The application doing its work, counting down.**

Fire up **android** to get the SDK/AVD manager, and then use the Virtual Devices tab to create and manage your virtual devices for testing (you can also use the command line). You can have as many AVDs as you like, to test on different setups, but for a basic one simply hit New, give it a name and choose the 2.2 target, leaving the other options as default. When the AVD is generated, hit Start.

Once it's running, you can install the application using the following:

```
adb install bin/CountdownActivity-debug.apk
```

and **adb** will automatically pick up the emulator.

Once installed, click the Launcher icon at the bottom of the phone screen, then the Countdown icon. The app will remain installed if you shut down the emulator and start it up again with the same AVD: all the existing information is stored as part of the AVD on shutdown.

Going further

You can probably already think of a few things to add to this basic application – playing an alarm at the end of the countdown would be one obvious option, or you could add a Stop button to the timer, and there's also the error pop-up message mentioned earlier. Mess around a bit with the code and see what else you can do with it, or try adding a text box to edit the text that shows at the end of the countdown.

You'll also notice that so far this program runs only on the virtual development device. In the next instalment of the series, we'll look at how to run it on a real phone, or even release it into the wilds of the Android Market. LXF

Quick tip

To reinstall a new version of the app, use **adb install -r bin/Countdown Activity-debug. apk**. Simple!

Next time!

Next time, we'll look at apps with more than one Activity and how they interact with one another; setting up menu items; slightly more complicated layouts; real phones and the Android Market; and a few more of the Android APIs.