Previous parts on the DVD

Android: Menus and ringtones

LINUX FORMAT On the DVD

Last month **Juliet Kemp** demonstrated how to set up, produce and test a basic countdown timer app. Now we're ready to make it do a little bit more...

Our expert

Juliet Kemp has spent a lot of time this month waiting for the Android emulator to fire up, so was particularly pleased to get her phone set up for testing instead.

This month we're going to expand the code we wrote in last month's tutorial to create a basic countdown timer, add a few more bells and whistles, and explore further what you can achieve with Android.

Even if you didn't read the last tutorial, you should still be able to follow this one if you have any familiarity with Java.

The code is on the coverdisc, so not all of it will feature in the article (Java can be pretty verbose at times). In particular, if you have compile issues, make sure you've imported all the necessary classes at the top of each source file.

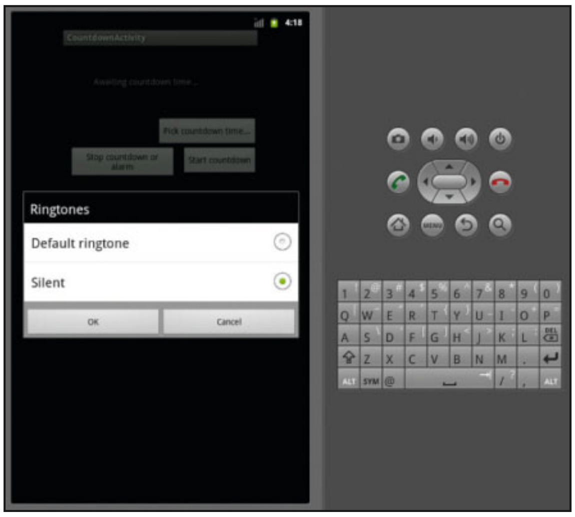
For more information on setting up an Android dev environment and on the basics of Android projects, check out either the previous tutorial in this series, or the (comprehensive) online Android docs – that's also the place to go for more detailed information about any of the classes and structures in this tutorial. Onwards!

What's on the menu?

The first thing we're going to do is add a menu, so that something happens when you click the Menu button. Later on we're going to add a ringtone that goes off when the countdown finishes, so the menu option will be to pick the ringtone you use.

For an options menu (a menu accessed by hitting the Menu button), there's no initial setup needed in the **onCreate** method; every Activity supports this sort of menu by default. Just add this method into the **CountdownActivity** class:

```
@Override public boolean onCreateOptionsMenu(Menu
menu) {
 super.onCreateOptionsMenu(menu);
 menu.add(Menu.NONE, SETTINGS_MENU_ID, Menu.
NONE, R.string.menu_settings);
 return true;
}
```



› The emulator's ringtone picker only has two options; if you test the app on your phone you'll see a wider range of ringtones.

The first line calls up to the method used by this class's superclass to create an options menu (which, helpfully, does most of the hard work for us). Then we add our single menu item. The first argument is a group ID (**NONE** here, which has the value 0, because we don't need to set up a group). The second is an identifier for the menu item, which we set as a constant at the top of the file:

```
private static final int SETTINGS_MENU_ID = Menu.FIRST;
```

As with **Menu.NONE**, this uses a constant (the first menu item) for portability and maintainability.

The third parameter sets the order for the item, which we don't set here, as we don't care about the order that our single item appears in. The final parameter is a string resource reference for the label of the item (see boxout). That's it; we now have a single menu item.

Next, we need a method to handle what happens when the item is selected; we want to fire up a ringtone picker:

```
@Override public boolean onMenuItemSelected(int id,
MenuItem item) {
 switch(item.getItemId()) {
  case SETTINGS_MENU_ID:
   chooseRingtone();
         return true;
  default:
   // we don't have any other menu items
 }
 return super.onMenuItemSelected(id, item);
}
```

Again, this is pretty straightforward, using a **switch** statement to look at the item ID and act accordingly. There's only one case here, as we have only one menu item, but it's good coding practice to set up a default and comment if (as here) it does nothing. Finally, the method that does the work:

```
private void chooseRingtone() {
 Intent i = new Intent(RingtoneManager.ACTION_
RINGTONE_PICKER);
 startActivityForResult(i, PICK_RINGTONE_REQUEST_ID);
}
```

See the boxout for more on actions and intents. **RingtoneManager** is one of the Android classes, and does what it says on the tin. The **ACTION_RINGTONE_PICKER** value is a static value that identifies what we want our new activity to do: pick a ringtone for this application.

But what do we do when the result comes back? This is handled by the **onActivityResult** method, which will be discussed in the next section; for now, we've fired the activity off with an intent and a request ID to identify it when it returns, and the first half of the menu is done.

You can, of course, add lots more menu items if you can think of things you want to do with them.

A step further...

In last issue's app, everything happened inside a single 'activity'. However, most Android apps have multiple activities, which talk to each other using intents. The boxout over the page explains how all of this fits together.

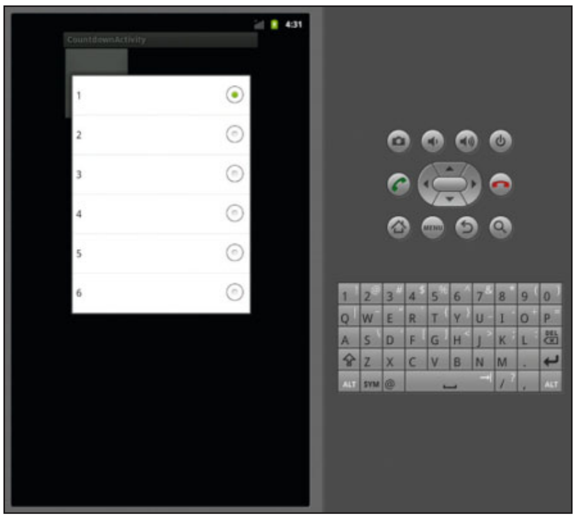
This time, instead of having a text box to enter the countdown time in, we're going to start a new activity to provide a scroller widget for the user to pick the number. To start this activity, there is a button that the user can click, and the code in **CountdownActivity** to set up that looks like this:

```
Button pickButton = (Button) findViewById(R.id.pickbutton);
pickButton.setOnClickListener(new View.OnClickListener() {
  @Override
   public void onClick(View view) {
    Intent i = new Intent(context, CountdownEnterTime.class);
    startActivityForResult(i, PICK_TIME_REQUEST_ID);
  }
});
```

As with the menu item we want to get a result back from the activity, so it needs both an intent and a request ID (set up at the top of the class). The intent identifies which class the activity to be called is in, and the ID allows us to work out which request we're getting a result back from in due course.

Before looking at how we deal with the activity result, let's create **CountdownEnterTime**, to generate that result. Check the code listing on the coverdisc for the full details of the class, as I won't list it all here, but the first part is the spinner to pick the number:

```
Spinner spinner = (Spinner) findViewById(R.id.spinner);
ArrayList<Integer> spinnerList = new ArrayList<Integer>();
for (int i = MIN; i <= MAX; i++) { spinnerList.add(i); }
ArrayAdapter<Integer> adapter = new ArrayAdapter(context,
    android.R.layout.simple_spinner_item, spinnerList);
adapter.setDropDownViewResource(
    android.R.layout.simple_spinner_dropdown_item);
spinner.setAdapter(adapter);
spinner.setOnItemSelectedListener(new
OnItemSelectedListener() {
```



› **Picking a seconds value from the spinner. Spinners can also take text data.**

Strings and resources

If you're referring to fixed strings in your Android code, it's best practice to use resource references to refer to them, rather than specifying them inline in the code. That way, if you want to change them it's easy to find them, and you're set up for localisation at a future time.

String resources are kept in the **res/values/strings.xml** file, and each entry in the file is an XML entity that looks like this:

```
<string name="countdown_
during">counting down:</string>
```

To refer to this string in your code, use **R.string.countdown_during**.

In fact, that resource reference isn't itself a string; instead, it's an integer that Android uses to locate the required string in the **strings.xml** file.

Most of the time, the substitution is seamless, so you can just treat a resource reference to a string as if it really were a string.

However, if you want to add two strings together, you'll need to do the conversion explicitly, as in the **onTick()** method in the **showTimer()** method:

```
countdownDisplay.setText(getString(R.
string.countdown_during) + seconds_
left);
```

Android resource reference strings also don't keep spaces at their ends (though they will in the middle). If you want a space at the end, use **\u0020** (the Unicode character for a space) at the end of the string, like this:

```
<string name="countdown_
during">counting down:\u0020</string>
```

```
  @Override
  public void onItemSelected(AdapterView<?> parent,
      View view, int pos, long id) {
    secondsPicked = (Integer)parent.getItemAtPosition(pos);
  }
  public void onNothingSelected(AdapterView parent) {
    // Do nothing.
  }
});
```

The spinner view is set up in the layout resources file (at **res/layout/picktime.xml**). A spinner needs an ArrayAdapter to translate between itself (displaying the data) and the backend array which holds that data. On this occasion, the data is an array of numbers (number of seconds to count down), so it's handled by an integer ArrayList. **android.R.layout.simple_spinner_item** is to stock Android layout resource that sets up a spinner layout for us; similarly with **android.R.layout.simple_spinner_dropdown_item**.

After connecting the Spinner, ArrayList, and ArrayAdapter together, the next step is to define what happens when an item is picked. All we want to do is store it in our class value **secondsPicked**. Note that if nothing is selected, the class does nothing; it's good practice always to comment if you're not doing anything with a method, just so that it's clear that you've done so deliberately.

After it's been picked, this value has to be passed back to the original activity, so it can be used for the countdown. For this, we set up a couple of buttons, OK and Cancel, whose **onClick** methods deal with packaging up the return info.

```
Button okButton = (Button) findViewById(R.id.okbutton);
Button cancelButton = (Button) findViewById(R.id.
cancelbutton);
okButton.setOnClickListener(new View.OnClickListener() {
  public void onClick(View view) {
    Intent i = new Intent();
    Bundle bundle = new Bundle();
    bundle.putInt(CountdownActivity.SECONDS_KEY,
secondsPicked);
    i.putExtras(bundle);
    setResult(RESULT_OK, i);
    finish();
  }
```

»

Quick tip

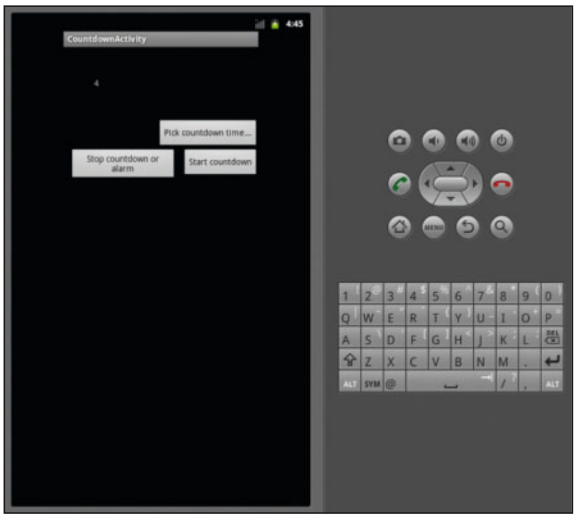
I've built this code against Android 2.2, but it should also compile on 2.3.* and even 3.0.

```
});
cancelButton.setOnClickListener(new View.
OnClickListener() {
  public void onClick(View view) {
    Intent i = new Intent();
    setResult(RESULT_CANCELED, i);
    finish();
  }
});
```

A bundle is used to store information in an intent, to be passed between activities. The OK button **onClick** method sets up an intent and a bundle, stores the **secondsPicked** value in the bundle, stores that in the intent, and sets the intent as the result of the activity. The Cancel button is more straightforward, as no **secondsPicked** value needs to be returned, so there's no bundle.

Meanwhile, back in CountdownActivity... we need to do something with this result that CountdownEnterTime has passed back in. To do this, we add this method at the end of the class:

```
@Override protected void onActivityResult(int requestCode,
int resultCode, Intent i) {
  super.onActivityResult(requestCode, resultCode, i);
  if (resultCode == RESULT_CANCELED) {
    return;
  }
  assert resultCode == RESULT_OK;
  switch(requestCode) {
    case PICK_TIME_REQUEST_ID:
      Bundle extras = i.getExtras();
      countdownMillis = extras.getInt(SECONDS_KEY);
      countdownDisplay.setText(Long.
toString(countdownSeconds));
      break;
    case PICK_RINGTONE_REQUEST_ID:
      Uri uri =
        i.getParcelableExtra(RingtoneManager.EXTRA_
RINGTONE_PICKED_URI);
      ringtone = RingtoneManager.getRingtone(context, uri);
      break;
    default:
      // do nothing; we don't expect any other results
    }
  }
}
```



› Ready to start the countdown. Note the 'stop' button as well as the 'start' button.

First, check for the **RESULT_CANCELLED** static value. This will be the same for any activity that's been called, and will always be treated the same: ignore and return. If we get past this stage, we assume that the result is OK; the assert statement makes this clear to anyone reading or maintaining the code. If any other return value comes back, the app will throw an error.

The switch statement means that this same method could deal with any one of numerous activities that might be called from this activity. We've got two: the time picker, and the ringtone picker. So, if it's the **PICK_TIME** ID that shows up, the value that was stored under **SECONDS_KEY** in the bundle is extracted.

The **countdownDisplay** line is just for ease-of-use; it shows the user the value that they've chosen, so they know that the right thing has happened. The other possible return value is from the ringtone picker. Ringtones are identified by a URI, and there's a static variable, **EXTRA_RINGTONE_PICKED_URI** that identifies the ringtone that was picked. The code extracts the URI from the bundle, then gets the ringtone and stores it in the **ringtone** member variable.

Now on to the actual timer...

Make some noise

The countdown timer itself is largely the same as in the last tutorial, using the Android-provided **CountDownTimer** class to change the text shown on every tick of the timer, and then show something at the end.

However, this time we're also going to make a noise when it rings. As discussed above in the menu, there's a RingtoneManager to deal with system ringtones, so again, a lot of the heavy lifting is done for us by the Android framework. The lines to add are in the **onFinish()** method within **showTimer()** (see the code on the coverdisc for the

Activities, intents and views

Android is designed to be modular, so that apps can hook into parts of other apps, minimising coding effort and component reuse.

Any app will have a default start point, but you can also jump into the app at other points by passing in the correct information (using an intent).

There are four basic component types:

» **Activities** provide a screen and user interface for a particular app or action. Most apps have multiple activities for their different aspects. Our *Countdown* app has an activity that runs the countdown itself, and another activity where the user picks the countdown time. The activity's visual content and user interface is provided by a 'view', which stores the layout of the app's UI.

» **Services** don't have a visual interface, but instead run in the background (eg playing music, or checking for email). A service will usually keep running after the user has navigated away from the application. Other components can bind to the service to interact with it.

» **Broadcast Receivers** receive and react to broadcast announcements, usually from the system itself. A broadcast receiver might start an activity in response to a particular system announcement.

» **Content Providers** are used to make some of the application's data available to other apps.

Activities, services, and broadcast receivers are activated by 'intents', which carry messages between components at run-time. The code in this tutorial uses intents to communicate between its two activities.

full listing):

```
public void onFinish() {
  countdownDisplay.setText(R.string.countdown_finish);
  if (ringtone == null) {
   ringtone = ringtoneMgr.getRingtone(context,
    android.provider.Settings.System.DEFAULT_RINGTONE_
    URI);
  }
  ringtone.play();
}
```

The text is set as before, with a resource reference. If the ringtone is null (for example, it hasn't been set via the menu item), it's set to be the current default ringtone (without this, the app would crash). **ringtone.play()** does exactly what you'd expect. All pleasingly straightforward.

There is, however, still something missing. The first time I tried this, I was pleased to get the ringtone, but less pleased when I realised I had no way of stopping it. So, let's hop back up to the **onCreate** method and set up a stop button:

```
Button stopButton = (Button) findViewById(R.id.stopbutton);
stopButton.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(View view) {
    if ( timer != null )   {
      timer.cancel();
      countdownDisplay.setText(Long.
toString(countdownSeconds));
    }
      if ( ringtone != null ) { ringtone.stop(); }
  }
});
```

In fact, while we're here, let's use the same button to stop either the timer mid-count, or the ringtone mid-ring. Since both of these are stored as member variables, and have their own stop/cancel methods, this is easy. It's important to check first that neither is null (as will be the case if the timer isn't currently counting-down, or the ringtone isn't currently ringing), but then we can just call the relevant methods. In the case of the timer, the countdown display is set back to its starting value, as well... Blissful silence. Lovely.

Time to get real

Before you release an app into the wild (aka the Android Market, on which more below), you should test it on at least one real hardware device. To do this, you can connect your phone via USB to your development box.

If using Ubuntu, there's a little setup to do so that your system will pick up your phone correctly, which will vary according to your device manufacturer; see **http://developer.android.com/guide/developing/device.html** for details on this. You'll also need to turn on USB Debugging on your device (under Menu > Settings > Applications > Development), and edit **AndroidManifest.xml** in your app so that the **<application>** looks like this:

```
<application [ ... ] android:debuggable="true">
```

Install the app to your phone with:

```
adb -d install -r bin/countdown-debug.apk
```

Note that you can get debugging info via **ddms** just as you can with an emulator.

So far, even when testing on your real device, you've only generated a debug version of your app. If you want to publish it, you'll need to produce a non-debug version.

Give your app a version number by setting **android.versionCode** and **android.versionName** in the **<manifest>** element in **AndroidManifest.xml**. You should also specify an icon in the **<application>** element, which you then put in **res/drawable**. Edit **android.label** to define the name that your app will have in the Applications menu, and remove **android:debuggable="true"**. Check, while you're at it, that you haven't left any debugging code lying around.

Next you need to generate a private key, using **keytool** (check that the link from **/usr/bin/keytool** points to a version within the JDK, not the **gjc** version). To publish in Android Market, the key must have a lifespan of at least 25 years (9,125 days).

```
keytool -genkey -v -keystore ~/android/release-keys.keystore
-alias mykey -keyalg RSA -validity 10000
```

ant release will generate an unsigned release version of your code, which you need to sign with this key, using *Jarsigner*. Next, you need to sign the release-compiled app, using *Jarsigner*:

```
jarsigner -verbose -keystore ~/android/release-keys.keystore
countdown-unsigned.apk mykey
```

mykey is the key alias as set up above, **countdown-unsigned.apk** the app itself, and **-keystore** gives the full path to the keystore. Verify the signing with

```
jarsigner -verify countdown.apk
```

Run another test of this final version, and then you can publish, either via your own website for manual download and install, or via Android Market, which requires setting up a developer account and paying $25.

Explore more

There's still a lot more you can do even with this simple program – for starters, it still leaves a lot to be desired visually. Alternatively, you could start exploring some of the other Android API, such as the gyroscope or the GPS and Map services.

As Android is open source, you can even get hold of the source code and dig into that a bit to give yourself some ideas. Just go ahead and get your coding hands dirty, and see what you can come up with. LXF

Ringtones on the emulator

If you try to run this code on the emulator, you'll get an error, and running **ddms** (the debugger) to find the error will show you this:

```
MediaPlayerService: Couldn't open fd
for content://settings/system/ringtone
```

This is because there are no ringtones loaded on the emulator. To get rid of this, fire up **ddms** while the emulator is running, and go to Device > File Explorer. Go to **/mnt/sdcard/**, and hit the Push button to choose an MP3 file to load onto the SD card. You may find that the permissions for this are incorrectly set. In this case, open up a terminal, and type:

```
$ mksdcard -l e 512M mysdcard.img
$ emulator -avd MyAVD2.2 -sdcard
mysdcard.img
```

which will restart the emulator with the named AVD and your new SD card. You may at this point be able to use **ddms** to push the file on; if not (I still had problems with mine), go back to the shell and instead use:

```
$ adb push myfile.mp3 /sdcard/
```

Restart your emulator (on the command-line, as above), and open the Music application. Navigate to the song you just added, long-click on it, and choose Set As Ringtone. You'll now be able to run the app on the emulator without a crash. You may need to reset the ringtone every time you restart.

Quick tip

If you have problems with the emulator ringtone setup – or any other problems with the emulator – it's surprisingly easy to set up a real phone to do testing on via USB.

Next time!

Next time, we'll find out a bit more about graphics and how views work. We'll also have a play with the gyroscope – one of the fun parts of coding for a phone!