CHAPTER 12

Using System Services

Android offers a number of system services, usually obtained by `getSystemService()` from your `Activity`, `Service`, or other `Context`. These are your gateway to all sorts of capabilities, from settings to volume to WiFi. Throughout the course of this book and its companion, we have seen several of these system services. In this chapter, we will take a look at others that may be of value to you in building compelling Android applications.

Get Alarmed

A common question when doing Android development is "where do I set up `cron` jobs?"

The `cron` utility – popular in Linux – is a way of scheduling work to be done periodically. You teach `cron` what to run and when to run it (e.g., weekdays at noon), and `cron` takes care of the rest. Since Android has a Linux kernel at its heart, one might think that `cron` might literally be available.

While `cron` itself is not, Android does have a system service named `AlarmManager` which fills a similar role. You give it a `PendingIntent` and a time (and optional a period for repeating) and it will fire off the `Intent` as needed. By this mechanism, you can get a similar effect to `cron`.

There is one small catch, though: Android is designed to run on mobile devices, particularly ones powered by all-too-tiny batteries. If you want

your periodic tasks to be run even if the device is "asleep", you will need to take a fair number of extra steps, mostly stemming around the concept of the `WakeLock`.

Concept of WakeLocks

Most of the time in Android, you are developing code that will run while the user is actually using the device. Activities, for example, only really make sense when the device is fully awake and the user is tapping on the screen or keyboard.

Particularly with scheduled background tasks, though, you need to bear in mind that the device will eventually "go to sleep". In full sleep mode, the display, main CPU, and keyboard are all powered off, to maximize battery life. Only on a low-level system event, like an incoming phone call, will anything wake up.

Another thing that will partially wake up the phone is an `Intent` raised by the `AlarmManager`. So long as broadcast receivers are processing that `Intent`, the `AlarmManager` ensures the CPU will be running (though the screen and keyboard are still off). Once the broadcast receivers are done, the `AlarmManager` lets the device go back to sleep.

You can achieve the same effect in your code via a `WakeLock`, obtained via the `PowerManager` system service. When you acquire a "partial `WakeLock`" (`PARTIAL_WAKE_LOCK`), you prevent the CPU from going back to sleep until you release said `WakeLock`. By proper use of a partial `WakeLock`, you can ensure the CPU will not get shut off while you are trying to do background work, while still allowing the device to sleep most of the time, in between alarm events.

However, using a `WakeLock` is a bit tricky, particularly when responding to an alarm `Intent`, as we will see in the next few sections.

Scheduling Alarms

The first step to creating a `cron` workalike is to arrange to get control when the device boots. After all, the `cron` daemon starts on boot as well, and we have no other way of ensuring that our background tasks start firing after a phone is reset.

We saw how to do that in a previous chapter – set up an `RECEIVE_BOOT_COMPLETED BroadcastReceiver`, with appropriate permissions. Here, for example, is the `AndroidManifest.xml` from `SystemServices/Alarm`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.commonsware.android.syssvc.alarm"
      android:versionCode="1"
      android:versionName="1.0">
  <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
  <uses-permission android:name="android.permission.WAKE_LOCK" />
  <application android:label="@string/app_name">
      <receiver android:name=".OnBootReceiver">
          <intent-filter>
              <action android:name="android.intent.action.BOOT_COMPLETED" />
          </intent-filter>
      </receiver>
      <receiver android:name=".OnAlarmReceiver">
      </receiver>
      <service android:name=".AppService">
      </service>
  </application>
</manifest>
```

We ask for an `OnBootReceiver` to get control when the device starts up, and it is in `OnBootReceiver` that we schedule our recurring alarm:

```
package com.commonsware.android.syssvc.alarm;

import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.SystemClock;
import android.util.Log;

public class OnBootReceiver extends BroadcastReceiver {
  private static final int PERIOD=300000;  // 5 minutes
```

```
  @Override
  public void onReceive(Context context, Intent intent) {
    AlarmManager
mgr=(AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
    Intent i=new Intent(context, OnAlarmReceiver.class);
    PendingIntent pi=PendingIntent.getBroadcast(context, 0,
                                          i, 0);

    mgr.setRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
                     SystemClock.elapsedRealtime(),
                     PERIOD,
                     pi);
  }
}
```

We get the `AlarmManager` via `getSystemService()`, create an `Intent` referencing another `BroadcastReceiver` (`OnAlarmReceiver`), wrap that `Intent` in a `PendingIntent`, and tell the `AlarmManager` to set up a repeating alarm via `setRepeating()`. By saying we want a `ELAPSED_REALTIME_WAKEUP` alarm, we indicate that we want the alarm to wake up the device (even if it is asleep) and to express all times using the time base used by `SystemClock.elapsedRealtime()`. In this case, our alarm is set to go off every five minutes.

This will cause the `AlarmManager` to raise our `Intent` imminently, and every five minutes thereafter.

Arranging for Work From Alarms

When an alarm goes off, our `OnAlarmReceiver` will get control. It needs to arrange for a service (in this case, named `AppService`) to do its work in the background, but then release control quickly – `onReceive()` cannot take very much time.

Here is the tiny implementation of `OnAlarmReceiver` from `SystemServices/Alarm`:

```
package com.commonsware.android.syssvc.alarm;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
```

```
import android.util.Log;

public class OnAlarmReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    WakefulIntentService.acquireStaticLock(context);

    context.startService(new Intent(context, AppService.class));
  }
}
```

While there is very little code in this class, it is merely deceptively simple.

First, we acquire a `WakeLock` from our `AppService`'s parent class, `WakefulIntentService` via `acquireStaticLock()`, shown below:

```
public static void acquireStaticLock(Context context) {
  getLock(context).acquire();
}

synchronized private static PowerManager.WakeLock getLock(Context context) {
  if (lockStatic==null) {
    PowerManager
mgr=(PowerManager)context.getSystemService(Context.POWER_SERVICE);

    lockStatic=mgr.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
                               LOCK_NAME_STATIC);
    lockStatic.setReferenceCounted(true);
  }

  return(lockStatic);
}
```

The `getLock()` implementation lazy-creates our `WakeLock` by getting the `PowerManager`, creating a new partial `WakeLock`, and setting it to be reference counted (meaning if it is acquired several times, it takes a corresponding number of `release()` calls to truly release the lock). If we have already retrieved the `WakeLock` in a previous invocation, we reuse the same lock.

Back in `OnAlarmReceiver`, up until this point, the CPU was running because `AlarmManager` held a partial `WakeLock`. Now, the CPU is running because both `AlarmManager` *and* `WakefulIntentService` hold a partial `WakeLock`.

Then, `OnAlarmReceiver` starts the `AppService` instance (remember: `acquireStaticLock()` was a *static* method) and exits. Notably,

`OnAlarmReceiver` does not release the `WakeLock` it acquired. This is important, as we need to ensure that the service can get its work done while the CPU is running. Had we released the `WakeLock` before returning, it is possible that the device would fall back asleep before our service had a chance to acquire a fresh `WakeLock`. This is one of the keys of using `WakeLock` successfully – as needed, use overlapping `WakeLock` instances to ensure constant coverage as you pass from component to component.

Now, our service will start up and be able to do something, while the CPU is running due to our acquired `WakeLock`.

Staying Awake At Work

So, `AppService` will now get control, under an active `WakeLock`. At minimum, our service will be called via `onStart()`, and possibly also `onCreate()` if the service had been previously stopped. Our mission is to do our work and release the `WakeLock`.

Since services should not do long-running tasks in `onStart()`, we could fork a `Thread`, have it do the work in the background, then have it release the `WakeLock`. Note that we cannot release the `WakeLock` in `onStart()` in this case – just because we have a background thread does not mean the device will keep the CPU running.

There are issues with forking a `Thread` for every incoming request, though:

- If the work needed to be done sometimes takes longer than the alarm period, we could wind up with many background threads, which is inefficient. It also means our `WakeLock` management gets much trickier, since we will not have released the `WakeLock` before the alarm tries to `acquire()` it again.
- If we also are invoked in `onStart()` via some foreground activity, we might wind up with many more bits of work to be done, again causing confusion with our `WakeLock` and perhaps slowing things down due to too many background threads.

Android has a class that helps with parts of this, `IntentService`. It arranges for a work queue of inbound `Intents` – rather than overriding `onStart()`, you override `onHandleIntent()`, which is called from a background thread. Android handles all the details of shutting down your service when there is no more outstanding work, managing the background thread, and so on.

However, `IntentService` does not do anything to hold a `WakeLock`.

Hence, this sample project implements `WakefulIntentService` as a subclass of `IntentService`. `WakefulIntentService` handles most of the `WakeLock` logic, so `AppService` (inheriting from `WakefulIntentService`) can just focus on the work it needs to do.

`WakefulIntentService` handles the `WakeLock` logic in four components:

1. It offers the public static method `acquireStaticLock()`, which needs to be called by whoever is calling `startService()` on our `WakefulIntentService` subclass.
2. In `onCreate()`, it creates (but does not acquire) another `WakeLock`. The static `WakeLock` will be used to keep the device awake while the `BroadcastReceiver` (or whoever else is calling `startService()`) starts up the service. The local `WakeLock` will be used to keep the device awake so long as there is work to be done.
3. In `onStart()`, it acquires the local `WakeLock`, lets the superclass do its work to enqueue the supplied Intent for later processing, then releases the static `WakeLock`. At this point, the device still must remain awake, because even though the `AlarmManager` `WakeLock` (used during the call to `onReceive()` in our `BroadcastReceiver`) is released, and our static `WakeLock` is released, our local `WakeLock` is still held.
4. In `onHandleIntent()`, it releases the local `WakeLock`. Since this `WakeLock` is reference-counted, the lock will only fully release once every `Intent` enqueued by `onStart()` has been handled by `onHandleIntent()`.

Here is the full implementation of `WakefulIntentService`:

```
package com.commonsware.android.syssvc.alarm;

import android.app.AlarmManager;
import android.app.PendingIntent;
import android.app.IntentService;
import android.content.Context;
import android.content.Intent;
import android.os.IBinder;
import android.os.PowerManager;
import android.util.Log;

public class WakefulIntentService extends IntentService {
  public static final String
LOCK_NAME_STATIC="com.commonsware.android.syssvc.AppService.Static";
  public static final String
LOCK_NAME_LOCAL="com.commonsware.android.syssvc.AppService.Local";
  private static PowerManager.WakeLock lockStatic=null;
  private PowerManager.WakeLock lockLocal=null;

  public static void acquireStaticLock(Context context) {
    getLock(context).acquire();
  }

  synchronized private static PowerManager.WakeLock getLock(Context context) {
    if (lockStatic==null) {
      PowerManager
mgr=(PowerManager)context.getSystemService(Context.POWER_SERVICE);

      lockStatic=mgr.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
                          LOCK_NAME_STATIC);
      lockStatic.setReferenceCounted(true);
    }

    return(lockStatic);
  }

  public WakefulIntentService(String name) {
    super(name);
  }

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

    PowerManager mgr=(PowerManager)getSystemService(Context.POWER_SERVICE);

    lockLocal=mgr.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
                            LOCK_NAME_LOCAL);
    lockLocal.setReferenceCounted(true);
  }

  @Override
  public void onStart(Intent intent, final int startId) {
    lockLocal.acquire();

    super.onStart(intent, startId);
```

```
    getLock(this).release();
  }

  @Override
  protected void onHandleIntent(Intent intent) {
    lockLocal.release();
  }
}
```

With all that behind us, `AppService` need only implement `onHandleIntent()`, do its work, and then chain upward to the `WakefulIntentService`'s implementation of `onHandleIntent()`:

```
package com.commonsware.android.syssvc.alarm;

import android.content.Intent;
import android.os.Environment;
import android.util.Log;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Date;

public class AppService extends WakefulIntentService {
  public AppService() {
    super("AppService");
  }

  @Override
  protected void onHandleIntent(Intent intent) {
    File log=new File(Environment.getExternalStorageDirectory(),
                      "AlarmLog.txt");

    try {
      BufferedWriter out=new BufferedWriter(new
FileWriter(log.getAbsolutePath(), true));

      out.write(new Date().toString());
      out.write("\n");
      out.close();
    }
    catch (IOException e) {
      Log.e("AppService", "Exception appending to log file", e);
    }

    super.onHandleIntent(intent);
  }
}
```

The "fake work" being done by this `AppService` is simply logging the fact that work needed to be done to a log file on the SD card.

Note that if you attempt to build and run this project that you will need an SD card in the device (or card image attached to your emulator).

Setting Expectations

If you have an Android device, you probably have spent some time in the Settings application, tweaking your device to work how you want – ringtones, WiFi settings, USB debugging, etc. Many of those settings are also available via `Settings` class (in the `android.provider` package), and particularly the `Settings.System` and `Settings.Secure` public inner classes.

Basic Settings

`Settings.System` allows you to get and, with the `WRITE_SETTINGS` permission, alter these settings. As one might expect, there are a series of typed getter and setter methods on `Settings.System`, each taking a key as a parameter. The keys are class constants, such as:

- `INSTALL_NON_MARKET_APPS` to control whether you can install applications on a device from outside of the Android Market
- `LOCK_PATTERN_ENABLED` to control whether the user needs to enter a lock pattern to enable use of the device
- `LOCK_PATTERN_VISIBLE` to control whether the lock pattern is drawn on-screen as it is swiped by the user, or if the swipes are "invisible"

The `SystemServices/Settings` project has a `SettingsSetter` sample application that displays a checklist:

```
<?xml version="1.0" encoding="utf-8"?>
<ListView xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@android:id/list"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
/>
```

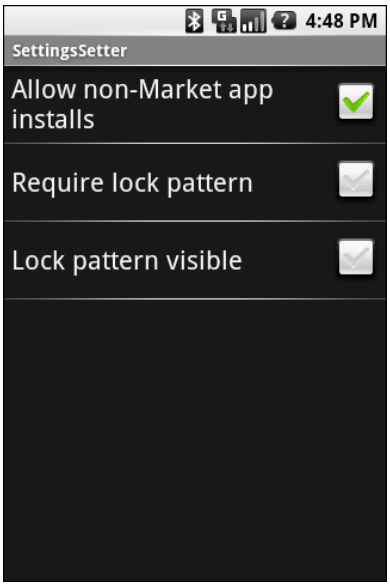

Figure 42. The SettingsSetter application

The checklist itself is filled with a few `BooleanSetting` objects, which map a display name with a `Settings.System` key:

```
static class BooleanSetting {
  String key;
  String displayName;

  BooleanSetting(String key, String displayName) {
    this.key=key;
    this.displayName=displayName;
  }

  @Override
  public String toString() {
    return(displayName);
  }

  boolean isChecked(ContentResolver cr) {
    try {
      int value=Settings.System.getInt(cr, key);

      return(value!=0);
    }
    catch (Settings.SettingNotFoundException e) {
      Log.e("SettingsSetter", e.getMessage());
    }

    return(false);
```

```
  }

  void setChecked(ContentResolver cr, boolean value) {
    Settings.System.putInt(cr, key, (value ? 1 : 0));
  }
}
```

Three such settings are put in the list, and as the checkboxes are checked and unchecked, the values are passed along to the settings themselves:

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

  getListView().setChoiceMode(ListView.CHOICE_MODE_MULTIPLE);
  setListAdapter(new ArrayAdapter(this,
                                  android.R.layout.simple_list_item_multiple_choi
ce,
                                  settings));

  ContentResolver cr=getContentResolver();

  for (int i=0;i<settings.size();i++) {
    BooleanSetting s=settings.get(i);

    getListView().setItemChecked(i, s.isChecked(cr));
  }
}

@Override
protected void onListItemClick(ListView l, View v,
                               int position, long id) {
  super.onListItemClick(l, v, position, id);

  BooleanSetting s=settings.get(position);

  s.setChecked(getContentResolver(),
               l.isItemChecked(position));
}
```

The `SettingsSetter` activity also has an option menu containing four items:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/app"
    android:title="Application"
    android:icon="@android:drawable/ic_menu_manage" />
  <item android:id="@+id/security"
    android:title="Security"
    android:icon="@android:drawable/ic_menu_close_clear_cancel" />
```

```
  <item android:id="@+id/wireless"
    android:title="Wireless"
    android:icon="@android:drawable/ic_menu_set_as" />
  <item android:id="@+id/all"
    android:title="All Settings"
    android:icon="@android:drawable/ic_menu_preferences" />
</menu>
```

These items correspond to four activity `Intent` values identified by the `Settings` class:

```
menuActivities.put(R.id.app,
                   Settings.ACTION_APPLICATION_SETTINGS);
menuActivities.put(R.id.security,
                   Settings.ACTION_SECURITY_SETTINGS);
menuActivities.put(R.id.wireless,
                   Settings.ACTION_WIRELESS_SETTINGS);
menuActivities.put(R.id.all,
                   Settings.ACTION_SETTINGS);
```

When an option menu is chosen, the corresponding activity is launched:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  String activity=menuActivities.get(item.getItemId());

  if (activity!=null) {
    startActivity(new Intent(activity));

    return(true);
  }

  return(super.onOptionsItemSelected(item));
}
```

This way, you have your choice of either directly manipulating the settings or merely making it easier for users to get to the Android-supplied activity for manipulating those settings.

Secure Settings

You will notice that if you use the above code and try changing the value of "Allow non-Market app installs", the change does not "stick" – once you exit and reopen the application, the setting returns to its original state.

Moreover, if you use the Settings application and examine the setting, it is clear that `SettingsSetter` is not actually changing that particular setting.

Once upon a time – Android 1.1 and earlier – it did.

Now, though, that setting is one that Android deems "secure". The constant has been moved from `Settings.System` to `Settings.Secure`, though the old constant is still there, flagged as deprecated.

These so-called "secure" settings are one that Android does not allow applications to change. No permission resolves this problem. The only option is to display the official Settings activity and let the user change the setting.

Can You Hear Me Now? OK, How About Now?

The fancier the device, the more complicated controlling sound volume becomes.

On a simple MP3 player, there is usually only one volume control. That is because there is only one source of sound: the music itself, played through speakers or headphones.

In Android, though, there are several sources of sounds:

- Ringing, to signify an incoming call
- Voice calls
- Alarms, such as those raised by the Alarm Clock application
- System sounds (error beeps, USB connection signal, etc.)
- Music, as might come from the MP3 player

Android allows the user to configure each of these volume levels separately. Usually, the user does this via the volume rocker buttons on the device, in the context of whatever sound is being played (e.g., when on a call, the

volume buttons change the voice call volume). Also, there is a screen in the Android Settings application that allows you to configure various volume levels.

The `AudioService` in Android allows you, the developer, to also control these volume levels, for all five "streams" (i.e., sources of sound). In the `SystemServices/Volume` project, we create a `Volumizer` application that displays and modifies all five volume levels, reusing the `Meter` widget we created in an earlier chapter.

Reusing Meter

Given that `Meter` was originally developed in a separate project, we had to do a few things to make it usable here.

First, we had to copy over the layout (`res/layout/meter.xml`), source (`src/com/commonsware/android/widget/Meter.java`), and two `Drawable` resources (`res/drawable/incr.png` and `res/drawable/decr.png`). We then moved it all into the same package as everything else (`com.commonsware.android.syssvc.volume`).

This, of course, defeats much of the reusability. Once better widget reuse models become apparent, expect updates to this book to cover them.

Attaching Meters to Volume Streams

Given that we have our `Meter` widget to work with, setting up `Meter` widgets to work with volume streams is fairly straightforward.

First, we need to create a layout with a `Meter` per stream:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res/com.commonsware.android.syssvc.v
olume"
  android:stretchColumns="1"
  android:layout_width="fill_parent"
```

```
  android:layout_height="fill_parent"
>
  <TableRow
    android:paddingTop="10px"
    android:paddingBottom="20px">
    <TextView android:text="Alarm:" />
    <com.commonsware.android.syssvc.volume.Meter
      android:id="@+id/alarm"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      app:incr="1"
      app:decr="1"
    />
  </TableRow>
  <TableRow
    android:paddingBottom="20px">
    <TextView android:text="Music:" />
    <com.commonsware.android.syssvc.volume.Meter
      android:id="@+id/music"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      app:incr="1"
      app:decr="1"
    />
  </TableRow>
  <TableRow
    android:paddingBottom="20px">
    <TextView android:text="Ring:" />
    <com.commonsware.android.syssvc.volume.Meter
      android:id="@+id/ring"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      app:incr="1"
      app:decr="1"
    />
  </TableRow>
  <TableRow
    android:paddingBottom="20px">
    <TextView android:text="System:" />
    <com.commonsware.android.syssvc.volume.Meter
      android:id="@+id/system"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      app:incr="1"
      app:decr="1"
    />
  </TableRow>
  <TableRow>
    <TextView android:text="Voice:" />
    <com.commonsware.android.syssvc.volume.Meter
      android:id="@+id/voice"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      app:incr="1"
```

```
      app:decr="1"
    />
  </TableRow>
</TableLayout>
```

Then, we need to wire up each of those meters in the `onCreate()` for `Volumizer`:

```
Meter alarm=null;
Meter music=null;
Meter ring=null;
Meter system=null;
Meter voice=null;
AudioManager mgr=null;

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

  mgr=(AudioManager)getSystemService(Context.AUDIO_SERVICE);

  alarm=(Meter)findViewById(R.id.alarm);
  music=(Meter)findViewById(R.id.music);
  ring=(Meter)findViewById(R.id.ring);
  system=(Meter)findViewById(R.id.system);
  voice=(Meter)findViewById(R.id.voice);

  alarm.setTag(AudioManager.STREAM_ALARM);
  music.setTag(AudioManager.STREAM_MUSIC);
  ring.setTag(AudioManager.STREAM_RING);
  system.setTag(AudioManager.STREAM_SYSTEM);
  voice.setTag(AudioManager.STREAM_VOICE_CALL);

  initMeter(alarm);
  initMeter(music);
  initMeter(ring);
  initMeter(system);
  initMeter(voice);
}
```

We use the tag for each `Meter` to hold the identifier for the stream associated with that specific `Meter`. That way, each `Meter` knows its stream.

In `initMeter()`, we set the appropriate size for the `Meter` bar via `setMax()`, set the initial value via `setProgress()`, and wire our increment and decrement events to the appropriate methods on `VolumeManager`:

```
    final int stream=((Integer)meter.getTag()).intValue();

    meter.setMax(mgr.getStreamMaxVolume(stream));
    meter.setProgress(mgr.getStreamVolume(stream));
    meter.setOnIncrListener(new View.OnClickListener() {
      public void onClick(View v) {
        mgr.adjustStreamVolume(stream,
                               AudioManager.ADJUST_RAISE, 0);
      }
    });
    meter.setOnDecrListener(new View.OnClickListener() {
      public void onClick(View v) {
        mgr.adjustStreamVolume(stream,
                               AudioManager.ADJUST_LOWER, 0);
      }
    });
  }
}
```

The net result is that when the user clicks the buttons on a meter, it adjusts the stream to match:

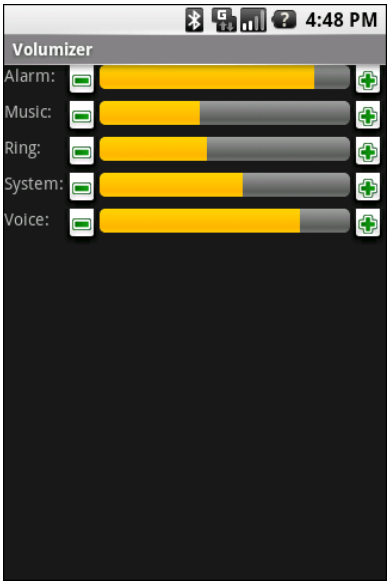

Figure 43. The Volumizer application

CHAPTER 13

Your Own (Advanced) Services

In *The Busy Coder's Guide to Android Development*, we covered how to create and consume services. Now, we can get into some more interesting facets of service implementations, notably remote services, so your service can serve activities outside of your application.

When IPC Attacks!

Services will tend to offer inter-process communication (IPC) as a means of interacting with activities or other Android components. Each service declares what methods it is making available over IPC; those methods are then available for other components to call, with Android handling all the messy details involved with making method calls across component or process boundaries.

The guts of this, from the standpoint of the developer, is expressed in AIDL: the Android Interface Description Language. If you have used IPC mechanisms like COM, CORBA, or the like, you will recognize the notion of IDL. AIDL describes the public IPC interface, and Android supplies tools to build the client and server side of that interface.

With that in mind, let's take a look at AIDL and IPC.

Write the AIDL

IDLs are frequently written in a "language-neutral" syntax. AIDL, on the other hand, looks a lot like a Java interface. For example, here is some AIDL:

```
package com.commonsware.android.advservice;

// Declare the interface.
interface IScript {
  void executeScript(String script);
}
```

As with a Java interface, you declare a package at the top. As with a Java interface, the methods are wrapped in an interface declaration (`interface IScript { ... }`). And, as with a Java interface, you list the methods you are making available.

The differences, though, are critical.

First, not every Java type can be used as a parameter. Your choices are:

- Primitive values (`int`, `float`, `double`, `boolean`, etc.)
- `String` and `CharSequence`
- `List` and `Map` (from `java.util`)
- Any other AIDL-defined interfaces
- Any Java classes that implement the `Parcelable` interface, which is Android's flavor of serialization

In the case of the latter two categories, you need to include `import` statements referencing the names of the classes or interfaces that you are using (e.g., `import com.commonsware.android.ISomething`). This is true even if these classes are in your own package – you have to import them anyway.

Next, parameters can be classified as `in`, `out`, or `inout`. Values that are `out` or `inout` can be changed by the service and those changes will be propagated

back to the client. Primitives (e.g., `int`) can only be `in`; we included in for the AIDL for `enable()` just for illustration purposes.

Also, you cannot throw any exceptions. You will need to catch all exceptions in your code, deal with them, and return failure indications some other way (e.g., error code return values).

Name your AIDL files with the `.aidl` extension and place them in the proper directory based on the package name.

When you build your project, either via an IDE or via Ant, the `aidl` utility from the Android SDK will translate your AIDL into a server stub and a client proxy.

Implement the Interface

Given the AIDL-created server stub, now you need to implement the service, either directly in the stub, or by routing the stub implementation to other methods you have already written.

The mechanics of this are fairly straightforward:

- Create a private instance of the AIDL-generated `.Stub` class (e.g., `IScript.Stub`)
- Implement methods matching up with each of the methods you placed in the AIDL
- Return this private instance from your `onBind()` method in the `Service` subclass

Note that AIDL IPC calls are synchronous, and so the caller is blocked until the IPC method returns. Hence, your services need to be quick about their work.

We will see examples of service stubs later in this chapter.

A Consumer Economy

Of course, we need to have a client for AIDL-defined services, lest these services feel lonely.

Bound for Success

To use an AIDL-defined service, you first need to create an instance of your own `ServiceConnection` class. `ServiceConnection`, as the name suggests, represents your connection to the service for the purposes of making IPC calls.

Your `ServiceConnection` subclass needs to implement two methods:

1. `onServiceConnected()`, which is called once your activity is bound to the service
2. `onServiceDisconnected()`, which is called if your connection ends normally, such as you unbinding your activity from the service

Each of those methods receives a `ComponentName`, which simply identifies the service you connected to. More importantly, `onServiceConnected()` receives an `IBinder` instance, which is your gateway to the IPC interface. You will want to convert the `IBinder` into an instance of your AIDL interface class, so you can use IPC as if you were calling regular methods on a regular Java class (`IScript.Stub.asInterface(binder)`).

To actually hook your activity to the service, call `bindService()` on the activity:

```
bindService(new Intent(IScript.class.getName()),
            svcConn, Context.BIND_AUTO_CREATE);
```

The `bindService()` method takes three parameters:

1. An `Intent` representing the service you wish to invoke
2. Your `ServiceConnection` instance

3. A set of flags – most times, you will want to pass in `BIND_AUTO_CREATE`, which will start up the service if it is not already running

After your `bindService()` call, your `onServiceConnected()` callback in the `ServiceConnection` will eventually be invoked, at which time your connection is ready for use.

Request for Service

Once your service interface object is ready (`IScript.Stub.asInterface(binder)`), you can start calling methods on it as you need to. In fact, if you disabled some widgets awaiting the connection, now is a fine time to re-enable them.

However, you will want to trap two exceptions. One is `DeadObjectException` – if this is raised, your service connection terminated unexpectedly. In this case, you should unwind your use of the service, perhaps by calling `onServiceDisconnected()` manually, as shown above. The other is `RemoteException`, which is a more general-purpose exception indicating a cross-process communications problem. Again, you should probably cease your use of the service.

Prometheus Unbound

When you are done with the IPC interface, call `unbindService()`, passing in the `ServiceConnection`. Eventually, your connection's `onServiceDisconnected()` callback will be invoked, at which point you should null out your interface object, disable relevant widgets, or otherwise flag yourself as no longer being able to use the service.

For example, in the WeatherPlus implementation of `onServiceDisconnected()` shown above, we null out the `IWeather` service object.

You can always reconnect to the service, via `bindService()`, if you need to use it again.

Service From Afar

Everything from the preceding two sections could be used by local services. In fact, that prose originally appeared in *The Busy Coder's Guide to Android Development* specifically in the context of local services. However, AIDL adds a fair bit of overhead, which is not necessary with local services. After all, AIDL is designed to marshal its parameters and transport them across process boundaries, which is why there are so many quirky rules about what you can and cannot pass as parameters to your AIDL-defined APIs.

So, given our AIDL description, let us examine some implementations, specifically for remote services.

Our sample applications – shown in the `AdvServices/RemoteService` and `AdvServices/RemoteClient` sample projects – convert our Beanshell demo from *The Busy Coder's Guide to Android Development* into a remote service. If you actually wanted to use scripting in an Android application, with scripts loaded off of the Internet, isolating their execution into a service might not be a bad idea. In the service, those scripts are sandboxed, only able to access files and APIs available to that service. The scripts cannot access your own application's databases, for example. If the script-executing service is kept tightly controlled, it minimizes the mischief a rogue script could possibly do.

Service Names

To bind to a service's AIDL-defined API, you need to craft an Intent that can identify the service in question. In the case of a local service, that Intent can use the local approach of directly referencing the service class.

Obviously, that is not possible in a remote service case, where the service class is not in the same process, and may not even be known by name to the client.

When you define a service to be used by remote, you need to add an intent-filter element to your service declaration in the manifest, indicating how you want that service to be referred to by clients. The manifest for `RemoteService` is shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.commonsware.android.advservice"
      android:versionCode="1"
      android:versionName="1.0">
  <application android:label="@string/app_name">
    <service android:name=".BshService">
      <intent-filter>
        <action android:name="com.commonsware.android.advservice.IScript" />
      </intent-filter>
    </service>
  </application>
</manifest>
```

Here, we say that the service can be identified by the name `com.commonsware.android.advservice.IScript`. So long as the client uses this name to identify the service, it can bind to that service's API.

In this case, the name is not an implementation, but the AIDL API, as you will see below. In effect, this means that so long as some service exists on the device that implements this API, the client will be able to bind to something.

The Service

Beyond the manifest, the service implementation is not too unusual. There is the AIDL interface, `IScript`:

```
package com.commonsware.android.advservice;

// Declare the interface.
interface IScript {
  void executeScript(String script);
}
```

And there is the actual service class itself, `BshService`:

```
package com.commonsware.android.advservice;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;
import bsh.Interpreter;

public class BshService extends Service {
  private final IScript.Stub binder=new IScript.Stub() {
    public void executeScript(String script) {
      executeScriptImpl(script);
    }
  };
  private Interpreter i=new Interpreter();

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

    try {
      i.set("context", this);
    }
    catch (bsh.EvalError e) {
      Log.e("BshService", "Error executing script", e);
    }
  }

  @Override
  public IBinder onBind(Intent intent) {
    return(binder);
  }

  @Override
  public void onDestroy() {
    super.onDestroy();
  }

  private void executeScriptImpl(String script) {
    try {
      i.eval(script);
    }
    catch (bsh.EvalError e) {
      Log.e("BshService", "Error executing script", e);
    }
  }
}
```

If you have seen the service and Beanshell samples in then this implementation will seem familiar. The biggest thing to note is that the service returns no result and handles any errors locally. Hence, the client will not get any response back from the script – the script will just run. In a

real implementation, this would be silly, and we will work to rectify this later in this chapter.

Also note that, in this implementation, the script is executed directly by the service on the calling thread. One might think this is not a problem, since the service is in its own process and, therefore, cannot possibly be using the client's UI thread. However, AIDL IPC calls are synchronous, so the client will still block waiting for the script to be executed. This too will be corrected later in this chapter.

The Client

The client – `BshServiceDemo` out of `AdvServices/RemoteClient` – is a fairly straight-forward mashup of the service and Beanshell clients, with two twists:

```
package com.commonsware.android.advservice.client;

import android.app.Activity;
import android.app.AlertDialog;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.Bundle;
import android.os.IBinder;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import com.commonsware.android.advservice.IScript;

public class BshServiceDemo extends Activity {
  private IScript service=null;
  private ServiceConnection svcConn=new ServiceConnection() {
    public void onServiceConnected(ComponentName className,
                                   IBinder binder) {
      service=IScript.Stub.asInterface(binder);
    }

    public void onServiceDisconnected(ComponentName className) {
      service=null;
    }
  };

  @Override
  public void onCreate(Bundle icicle) {
```

```
    super.onCreate(icicle);
    setContentView(R.layout.main);

    Button btn=(Button)findViewById(R.id.eval);
    final EditText script=(EditText)findViewById(R.id.script);

    btn.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        String src=script.getText().toString();

        try {
          service.executeScript(src);
        }
        catch (android.os.RemoteException e) {
          AlertDialog.Builder builder=
                    new AlertDialog.Builder(BshServiceDemo.this);

          builder
            .setTitle("Exception!")
            .setMessage(e.toString())
            .setPositiveButton("OK", null)
            .show();
        }
      }
    });

    bindService(new Intent(IScript.class.getName()),
                svcConn, Context.BIND_AUTO_CREATE);
  }

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

    unbindService(svcConn);
  }
}
```

One twist is that the client needs its own copy of `IScript.aidl`. After all, it is a totally separate application, and therefore does not share source code with the service. In a production environment, we might craft and distribute a JAR file that contains the `IScript` classes, so both client and service can work off the same definition (see the upcoming chapter on reusable components). For now, we will just have a copy of the AIDL.

Then, the `bindService()` call uses a slightly different `Intent`, one that references the name of the AIDL interface's class implementation. That happens to be the name the service is registered under, and that is the glue that allows the client to find the matching service.

If you compile both applications and upload them to the device, then start up the client, you can enter in Beanshell code and have it be executed by the service. Note, though, that you cannot perform UI operations (e.g., raise a `Toast`) from the service. If you choose some script that is long-running, you will see that the Go! button is blocked until the script is complete:

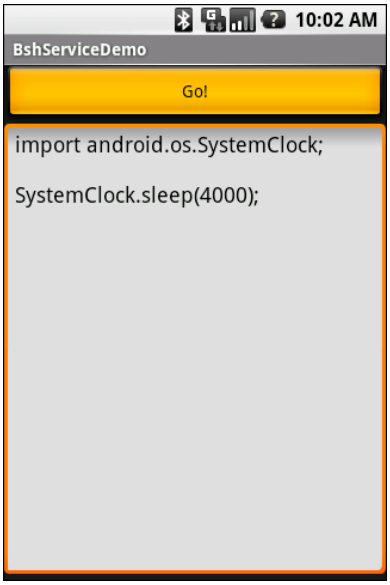

Figure 44. The BshServiceDemo application, running a long script

Servicing the Service

The preceding section outlined two flaws in the implementation of the Beanshell remote service:

1. The client received no results from the script execution
2. The client blocked waiting for the script to complete

If we were not worried about the blocking-call issue, we could simply have the `executeScript()` exported API return some sort of result (e.g., `toString()` on the result of the Beanshell `eval()` call). However, that would not solve the fact that calls to service APIs are synchronous even for remote services.

Another approach would be to pass some sort of callback object with `executeScript()`, such that the server could run the script asynchronously and invoke the callback on success or failure. This, though, implies that there is some way to have the activity export an API to the service.

Fortunately, this is eminently doable, as you will see in this section, and the accompanying samples (`AdvServices/RemoteServiceEx` and `AdvServices/RemoteClientEx`).

Callbacks via AIDL

AIDL does not have any concept of direction. It just knows interfaces and stub implementations. In the preceding example, we used AIDL to have the service flesh out the stub implementation and have the client access the service via the AIDL-defined interface. However, there is nothing magic about services implementing and clients accessing – it is equally possible to reverse matters and have the client implement something the service uses via an interface.

So, for example, we could create an `IScriptResult.aidl` file:

```
package com.commonsware.android.advservice;

// Declare the interface.
interface IScriptResult {
  void success(String result);
  void failure(String error);
}
```

Then, we can augment `IScript` itself, to pass an `IScriptResult` with `executeScript()`:

```
package com.commonsware.android.advservice;

import com.commonsware.android.advservice.IScriptResult;

// Declare the interface.
interface IScript {
  void executeScript(String script, IScriptResult cb);
}
```

Notice that we need to specifically import `IScriptResult`, just like we might import some "regular" Java interface. And, as before, we need to make sure the client and the server are working off of the same AIDL definitions, so these two AIDL files need to be replicated across each project.

But other than that one little twist, this is all that is required, at the AIDL level, to have the client pass a callback object to the service: define the AIDL for the callback and add it as a parameter to some service API call.

Of course, there is a little more work to do on the client and server side to make use of this callback object.

Revising the Client

On the client, we need to implement an `IScriptResult`. On `success()`, we can do something like raise a `Toast`; on `failure()`, we can perhaps show an `AlertDialog`.

The catch is that we cannot be certain we are being called on the UI thread in our callback object.

So, the safest way to do that is to make the callback object use something like `runOnUiThread()` to ensure the results are displayed on the UI thread:

```
private final IScriptResult.Stub callback=new IScriptResult.Stub() {
  public void success(final String result) {
    runOnUiThread(new Runnable() {
      public void run() {
        successImpl(result);
      }
    });
  }

  public void failure(final String error) {
    runOnUiThread(new Runnable() {
      public void run() {
        failureImpl(error);
      }
    });
  }
};
```

```
private void successImpl(String result) {
  Toast
    .makeText(BshServiceDemo.this, result, Toast.LENGTH_LONG)
    .show();
}

private void failureImpl(String error) {
  AlertDialog.Builder builder=
            new AlertDialog.Builder(BshServiceDemo.this);

  builder
    .setTitle("Exception!")
    .setMessage(error)
    .setPositiveButton("OK", null)
    .show();
}
```

And, of course, we need to update our call to `executeScript()` to pass the callback object to the remote service:

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

  Button btn=(Button)findViewById(R.id.eval);
  final EditText script=(EditText)findViewById(R.id.script);

  btn.setOnClickListener(new View.OnClickListener() {
    public void onClick(View view) {
      String src=script.getText().toString();

      try {
        service.executeScript(src, callback);
      }
      catch (android.os.RemoteException e) {
        failureImpl(e.toString());
      }
    }
  });

  bindService(new Intent(IScript.class.getName()),
              svcConn, Context.BIND_AUTO_CREATE);
}
```

Revising the Service

The service also needs changing, to both execute the scripts asynchronously and use the supplied callback object for the end results of the script's execution.

As was demonstrated in the chapter on `Camera`, `BshService` from `AdvServices/RemoteServiceEx` uses the `LinkedBlockingQueue` pattern to manage a background thread. An `ExecuteScriptJob` wraps up the script and callback; when the job is eventually processed, it uses the callback to supply the results of the `eval()` (on success) or the message of the `Exception` (on failure):

```
package com.commonsware.android.advservice;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;
import java.util.concurrent.LinkedBlockingQueue;
import bsh.Interpreter;

public class BshService extends Service {
  private final IScript.Stub binder=new IScript.Stub() {
    public void executeScript(String script, IScriptResult cb) {
      executeScriptImpl(script, cb);
    }
  };
  private Interpreter i=new Interpreter();
  private LinkedBlockingQueue<Job> q=new LinkedBlockingQueue<Job>();

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

    new Thread(qProcessor).start();

    try {
      i.set("context", this);
    }
    catch (bsh.EvalError e) {
      Log.e("BshService", "Error executing script", e);
    }
  }

  @Override
  public IBinder onBind(Intent intent) {
    return(binder);
```

```
  }

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

    q.add(new KillJob());
  }

  private void executeScriptImpl(String script,
                                 IScriptResult cb) {
    q.add(new ExecuteScriptJob(script, cb));
  }

  Runnable qProcessor=new Runnable() {
    public void run() {
      while (true) {
        try {
          Job j=q.take();

          if (j.stopThread()) {
            break;
          }
          else {
            j.process();
          }
        }
        catch (InterruptedException e) {
          break;
        }
      }
    }
  };

  class Job {
    boolean stopThread() {
      return(false);
    }

    void process() {
      // no-op
    }
  }

  class KillJob extends Job {
    @Override
    boolean stopThread() {
      return(true);
    }
  }

  class ExecuteScriptJob extends Job {
    IScriptResult cb;
    String script;
```

```
    ExecuteScriptJob(String script, IScriptResult cb) {
      this.script=script;
      this.cb=cb;
    }

    void process() {
      try {
        cb.success(i.eval(script).toString());
      }
      catch (Throwable e) {
        Log.e("BshService", "Error executing script", e);

        try {
          cb.failure(e.getMessage());
        }
        catch (Throwable t) {
          Log.e("BshService",
                "Error returning exception to client",
                t);
        }
      }
    }
  }
}
```

Notice that the service's own API just needs the `IScriptResult` parameter, which can be passed around and used like any other Java object. The fact that it happens to cause calls to be made synchronously back to the remote client is invisible to the service.

The net result is that the client can call the service and get its results without tying up the client's UI thread.

CHAPTER 14

Finding Available Actions via Introspection

Sometimes, you know just what you want to do, such as display one of your other activities.

Sometimes, you have a pretty good idea of what you want to do, such as view the content represented by a `Uri`, or have the user pick a piece of content of some MIME type.

Sometimes, you're lost. All you have is a content `Uri`, and you don't really know what you can do with it.

For example, suppose you were creating a common tagging subsystem for Android, where users could tag pieces of content – contacts, Web URLs, geographic locations, etc. Your subsystem would hold onto the `Uri` of the content plus the associated tags, so other subsystems could, say, ask for all pieces of content referencing some tag.

That's all well and good. However, you probably need some sort of maintenance activity, where users could view all their tags and the pieces of content so tagged. This might even serve as a quasi-bookmark service for items on their phone. The problem is, the user is going to expect to be able to do useful things with the content they find in your subsystem, such as dial a contact or show a map for a location.

The problem is, you have absolutely no idea what is all possible with any given content `Uri`. You probably can view any of them, but can you edit them? Can you dial them? Since new applications with new types of content could be added by any user at any time, you can't even assume you know all possible combinations just by looking at the stock applications shipped on all Android devices.

Fortunately, the Android developers thought of this.

Android offers various means by which you can present to your users a set of likely activities to spawn for a given content `Uri`...even if you have no idea what that content `Uri` really represents. This chapter explores some of these `Uri` action introspection tools.

Pick 'Em

Sometimes, you know your content `Uri` represents a collection of some type, such as `content://contacts/people` representing the list of contacts in the stock Android contacts list. In this case, you can let the user pick a contact that your activity can then use (e.g., tag it, dial it).

To do this, you need to create an intent for the `ACTION_PICK` on the target `Uri`, then start a sub activity (via `startActivityForResult()`) to allow the user to pick a piece of content of the specified type. If your `onActivityResult()` callback for this request gets a `RESULT_OK` result code, your data string can be parsed into a `Uri` representing the chosen piece of content.

For example, take a look at `Introspection/Pick` in the sample applications. This activity gives you a field for a collection `Uri` (with `content://contacts/people` pre-filled in for your convenience), plus a really big “Gimme!” button:

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

```
  <EditText android:id="@+id/type"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:cursorVisible="true"
    android:editable="true"
    android:singleLine="true"
    android:text="content://contacts/people"
  />
  <Button
    android:id="@+id/pick"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:text="Gimme!"
    android:layout_weight="1"
  />
</LinearLayout>
```

Upon being clicked, the button creates the `ACTION_PICK` on the user-supplied collection `Uri` and starts the sub-activity. When that sub-activity completes with `RESULT_OK`, the `ACTION_VIEW` is invoked on the resulting content `Uri`.

```
public class PickDemo extends Activity {
  static final int PICK_REQUEST=1337;
  private EditText type;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    type=(EditText)findViewById(R.id.type);

    Button btn=(Button)findViewById(R.id.pick);

    btn.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        Intent i=new Intent(Intent.ACTION_PICK,
                    Uri.parse(type.getText().toString()));

        startActivityForResult(i, PICK_REQUEST);
      }
    });
  }

  @Override
  protected void onActivityResult(int requestCode, int resultCode,
                                  Intent data) {
    if (requestCode==PICK_REQUEST) {
      if (resultCode==RESULT_OK) {
          startActivity(new Intent(Intent.ACTION_VIEW,
                                  data.getData()));
```

```
      }
    }
  }
}
```

The result: the user chooses a collection, picks a piece of content, and views it.

Figure 45. The PickDemo sample application, as initially launched

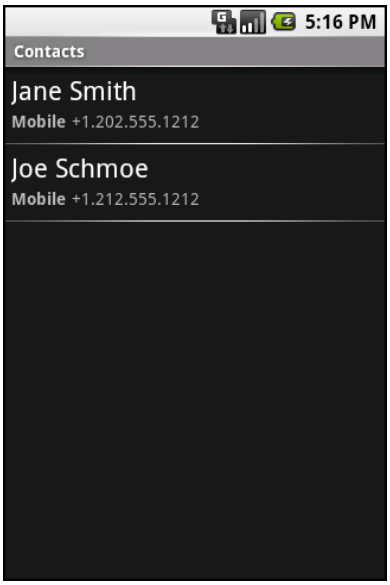

Figure 46. The same application, after clicking the "Gimme!" button, showing the list of available people

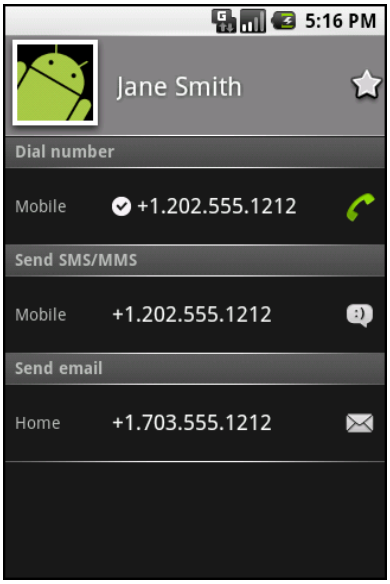

Figure 47. A view of a contact, launched by PickDemo after choosing one of the people from the pick list

Would You Like to See the Menu?

Another way to give the user ways to take actions on a piece of content, without you knowing what actions are possible, is to inject a set of menu choices into the options menu via `addIntentOptions()`. This method, available on `Menu`, takes an `Intent` and other parameters and fills in a set of menu choices on the `Menu` instance, each representing one possible action. Choosing one of those menu choices spawns the associated activity.

The canonical example of using `addIntentOptions()` illustrates another flavor of having a piece of content and not knowing the actions that can be taken. Android applications are perfectly capable of adding new actions to existing content types, so even though you wrote your application and know what you expect to be done with your content, there may be other options you are unaware of that are available to users.

For example, imagine the tagging subsystem mentioned in the introduction to this chapter. It would be very annoying to users if, every time they wanted to tag a piece of content, they had to go to a separate tagging tool, then turn around and pick the content they just had been working on (if that is even technically possible) before associating tags with it. Instead, they would probably prefer a menu choice in the content's own "home" activity where they can indicate they want to tag it, which leads them to the set-a-tag activity and tells that activity what content should get tagged.

To accomplish this, the tagging subsystem should set up an intent filter, supporting any piece of content, with their own action (e.g., `ACTION_TAG`) and a category of `CATEGORY_ALTERNATIVE`. The category `CATEGORY_ALTERNATIVE` is the convention for one application adding actions to another application's content.

If you want to write activities that are aware of possible add-ons like tagging, you should use `addIntentOptions()` to add those add-ons' actions to your options menu, such as the following:

```
Intent intent = new Intent(null, myContentUri);
```

```
intent.addCategory(Intent.ALTERNATIVE_CATEGORY);
menu.addIntentOptions(Menu.ALTERNATIVE, 0,
                      new ComponentName(this,
                                        MyActivity.class),
                      null, intent, 0, null);
```

Here, `myContentUri` is the content `Uri` of whatever is being viewed by the user in this activity, `MyActivity` is the name of the activity class, and `menu` is the menu being modified.

In this case, the `Intent` we are using to pick actions from requires that appropriate intent receivers support the `CATEGORY_ALTERNATIVE`. Then, we add the options to the menu with `addIntentOptions()` and the following parameters:

- The sort position for this set of menu choices, typically set to `0` (appear in the order added to the menu) or `ALTERNATIVE` (appear after other menu choices)
- A unique number for this set of menu choices, or `0` if you do not need a number
- A `ComponentName` instance representing the activity that is populating its menu – this is used to filter out the activity's own actions, so the activity can handle its own actions as it sees fit
- An array of `Intent` instances that are the “specific” matches – any actions matching those intents are shown first in the menu before any other possible actions
- The `Intent` for which you want the available actions
- A set of flags. The only one of likely relevance is represented as `MATCH_DEFAULT_ONLY`, which means matching actions must also implement the `DEFAULT_CATEGORY` category. If you do not need this, use a value of `0` for the flags.
- An array of `Menu.Item`, which will hold the menu items matching the array of `Intent` instances supplied as the “specifics”, or `null` if you do not need those items (or are not using “specifics”)

Asking Around

The `addIntentOptions()` method in turn uses `queryIntentActivityOptions()` for the "heavy lifting" of finding possible actions. The `queryIntentActivityOptions()` method is implemented on `PackageManager`, which is available to your activity via `getPackageManager()`.

The `queryIntentActivityOptions()` method takes some of the same parameters as does `addIntentOptions()`, notably the caller `ComponentName`, the "specifics" array of `Intent` instances, the overall `Intent` representing the actions you are seeking, and the set of flags. It returns a `List` of `Intent` instances matching the stated criteria, with the "specifics" ones first.

If you would like to offer alternative actions to users, but by means other than `addIntentOptions()`, you could call `queryIntentActivityOptions()`, get the `Intent` instances, then use them to populate some other user interface (e.g., a toolbar).