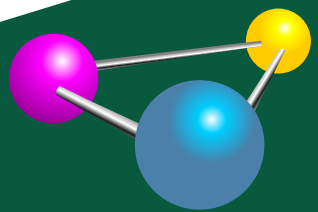
Version 0.9

digital updates at commonsware.com!

Android Programming Tutorials

Mark L. Murphy



CommonsWare

Android Programming Tutorials

by Mark L. Murphy

Android Programming Tutorials
by Mark L. Murphy

CommonsWare books may be purchased in printed (bulk) or digital form for educational or business use. For more information, contact *direct@commonsware.com*.

Printing History:
Apr 2009: Version 0.9 ISBN: 978-0-9816780-2-3

Table of Contents

Welcome to the Warescription!

We hope you enjoy this digital book and its updates – keep tabs on the Warescription feed off the CommonsWare site to learn when new editions of this book, or other books in your Warescription, are available.

Each Warescription digital book is licensed for the exclusive use of its subscriber and is tagged with the subscribers name. We ask that you not distribute these books. If you work for a firm and wish to have several employees have access, enterprise Warescriptions are available. Just contact us at enterprise@commonsware.com.

Also, bear in mind that eventually this edition of this title will be released under a Creative Commons license – more on this in the preface.

Remember that the CommonsWare Web site has errata and resources (e.g., source code) for each of our titles. Just visit the Web page for the book you are interested in and follow the links.

Some notes for Kindle users:

- You may wish to drop your font size to level 2 for easier reading
- Source code listings are incorporated as graphics so as to retain the monospace font, though this means the source code listings do not honor changes in Kindle font size

Preface

Welcome to the Book!

If you come to this book after having read its companion volumes, The Busy Coder's Guide to Android Development and The Busy Coder's Guide to Advanced Android Development, thanks for sticking with the series! CommonsWare aims to have the most comprehensive set of Android development resources (outside of the Open Handset Alliance itself), and we appreciate your interest.

If you come to this book having learned about Android from other sources, thanks for joining the CommonsWare community!

Prerequisites

This book is a collection of tutorials, walking you through developing Android applications, from the simplest "Hello, world!" to applications using many advanced Android APIs.

Since this book only supplies tutorials, you will want something beyond it as a reference guide. That could be simply the Android SDK documentation, available with your SDK installation or online. It could be the other books in the CommonsWare Android series. Or, it could be another Android book – a list of currently-available Android books can be found on the Android Programming knol. What you do not want to do is attempt to learn all of

Android solely from these tutorials, as they will demonstrate the breadth of the Android API but not its depth.

Also, the tutorials themselves have varying depth. Early on, there is more "hand-holding" to explain every bit of what needs to be done (e.g., classes to import). As the tutorials progress, some of the simpler Java bookkeeping steps are left out of the instructions, so the tutorials can focus on the Android aspects of the code.

Using the Tutorials

Each tutorial has a main set of step-by-step instructions, plus an "Extra Credit" section. The step-by-step instructions are intended to guide you through creating or extending Android applications, including all code you need to enter and all commands you need to run. The "Extra Credit" sections, on the other hand, provide some suggested areas for experimentation beyond the base tutorial, without step-by-step instructions.

If you only wish to do the "Extra Credit" work, you can download the results of each tutorial's step-by-step instructions from the CommonsWare site.

The tutorials assume you are not necessarily using Eclipse, let alone any other specific editor or debugger. The instructions included in the tutorials will speak in general terms when it comes to tools outside of those supplied by the Android SDK itself.

The tutorials include instructions for both Linux and Windows XP. OS X developers should be able to follow the Linux instructions in general, making slight alterations as needed for your platform. Windows Vista users should be able to follow the Windows XP instructions in general, tweaking the steps to deal with Vista's directory structure and revised Start menu.

Warescription

This book will be published both in print and in digital form. The digital versions of all CommonsWare titles are available via an annual subscription – the Warescription.

The Warescription entitles you, for the duration of your subscription, to digital forms of *all* CommonsWare titles, not just the one you are reading. Presently, CommonsWare offers PDF and Kindle; other digital formats will be added based on interest and the openness of the format.

Each subscriber gets personalized editions of all editions of each title: both those mirroring printed editions and in-between updates that are only available in digital form. That way, your digital books are never out of date for long, and you can take advantage of new material as it is made available instead of having to wait for a whole new print edition. For example, when new releases of the Android SDK are made available, this book will be quickly updated to be accurate with changes in the APIs.

From time to time, subscribers will also receive access to subscriber-only online material, including not-yet-published new titles.

Also, if you own a print copy of a CommonsWare book, and it is in good clean condition with no marks or stickers, you can exchange that copy for a free four-month Warescription.

If you are interested in a Warescription, visit the Warescription section of the CommonsWare Web site.

Book Bug Bounty

Find a problem in one of our books? Let us know!

Be the first to report a unique concrete problem in the current digital edition, and we'll give you a coupon for a six-month Warescription as a bounty for helping us deliver a better product. You can use that coupon to get a new Warescription, renew an existing Warescription, or give the

coupon to a friend, colleague, or some random person you meet on the subway.

By "concrete" problem, we mean things like:

- Typographical errors
- Sample applications that do not work as advertised, in the environment described in the book
- Factual errors that cannot be open to interpretation

By "unique", we mean ones not yet reported. Each book has an errata page on the CommonsWare Web site; most known problems will be listed there. One coupon is given per email containing valid bug reports.

NOTE: Books with version numbers lower than 0.9 are ineligible for the bounty program, as they are in various stages of completion. We appreciate bug reports, though, if you choose to share them with us.

We appreciate hearing about "softer" issues as well, such as:

- Places where you think we are in error, but where we feel our interpretation is reasonable
- Places where you think we could add sample applications, or expand upon the existing material
- Samples that do not work due to "shifting sands" of the underlying environment (e.g., changed APIs with new releases of an SDK)

However, those "softer" issues do not qualify for the formal bounty program.

Questions about the bug bounty, or problems you wish to report for bounty consideration, should be sent to bounty@commonsware.com.

Source Code License

The source code samples shown in this book are available for download from the CommonsWare Web site. All of the Android projects are licensed under the Apache 2.0 License, in case you have the desire to reuse any of it.

Creative Commons and the Four-to-Free (42F) Guarantee

Each CommonsWare book edition will be available for use under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 license as of the fourth anniversary of its publication date, or when 4,000 copies of the edition have been sold, whichever comes first. That means that, once four years have elapsed (perhaps sooner!), you can use this prose for non-commercial purposes. That is our Four-to-Free Guarantee to our readers and the broader community. For the purposes of this guarantee, new Warescriptions and renewals will be counted as sales of this edition, starting from the time the edition is published.

This edition of this book will be available under the aforementioned Creative Commons license on March 1, **2012**. Of course, watch the CommonsWare Web site, as this edition might be relicensed sooner based on sales.

For more details on the Creative Commons Attribution-Noncommercial-Share Alike 3.0 license, visit the Creative Commons Web site.

Note that future editions of this book will become free on later dates, each four years from the publication of that edition or based on sales of that specific edition. Releasing one edition under the Creative Commons license does not automatically release *all* editions under that license.

Lifecycle of a CommonsWare Book

CommonsWare books generally go through a series of stages.

First are the pre-release editions. These will have version numbers below 0.9 (e.g., 0.2). These editions are incomplete, often times having but a few chapters to go along with outlines and notes. However, we make them available to those on the Warescription so they can get early access to the material.

Release candidates are editions with version numbers ending in ".9" (0.9, 1.9, etc.). These editions should be complete. Once again, they are made available to those on the Warescription so they get early access to the material and can file bug reports (and receive bounties in return!).

Major editions are those with version numbers ending in ".0" (1.0, 2.0, etc.). These will be first published digitally for the Warescription members, but will shortly thereafter be available in print from booksellers worldwide.

Versions between a major edition and the next release candidate (e.g., 1.1, 1.2) will contain bug fixes plus new material. Each of these editions should also be complete, in that you will not see any "TBD" (to be done) markers or the like. However, these editions may have bugs, and so bug reports are eligible for the bounty program, as with release candidates and major releases.

A book usually will progress fairly rapidly through the pre-release editions to the first release candidate and Version 1.0 – often times, only a few months. Depending on the book's scope, it may go through another cycle of significant improvement (versions 1.1 through 2.0), though this may take several months to a year or more. Eventually, though, the book will go into more of a "maintenance mode", only getting updates to fix bugs and deal with major ecosystem events – for example, a new release of the Android SDK will necessitate an update to all Android books.

What's in the Book?

This book works through two main example applications:

- `LunchList`, which lets you track restaurants that you like to visit over lunch
- `Patchy`, a Twitter client

Here is what you will find in each of the tutorials:

- In Tutorial 1, you will create a quick sample application, to demonstrate how to install and use the Android SDK

- In Tutorial 2, you will create a skeletal edition of `LunchList`, starting with a form to collect details about a restaurant
- In Tutorial 3, you will add some radio buttons to the `LunchList` form
- In Tutorial 4, you will add support for multiple restaurants and show them in a list, alongside the restaurant details form
- In Tutorial 5, you will make the list look a bit better by adding custom formatting to the list rows
- In Tutorial 6, you will split the list and detail form onto separate tabs, to give each more screen space
- In Tutorial 7, you will add an option menu to the application, along with a menu choice displaying a brief message in a popup
- In Tutorial 8, you will add a progress bar and have it be updated by a background thread
- In Tutorial 9, we will ensure our progress bar and background thread continue working even if something else takes over the device, such as an incoming call
- In Tutorial 10, we will add support for a landscape layout, so our application looks fine no matter how the screen is oriented
- In Tutorial 11, we will start storing our restaurant information in a SQLite database
- In Tutorial 12, we will split the list of restaurants and the restaurant detail form into separate screens ("activities" in Android terms)
- In Tutorial 13, we will add in support for user preferences, allowing the user to choose the sort order of the restaurants in the list
- In Tutorial 14, we will add more support for screen rotations, to save our state when the device's keyboard is opened or otherwise changed between portrait and landscape orientations
- In Tutorial 15, we will ask for permission to place phone calls to restaurants
- In Tutorial 16, we will actually place those calls, plus allow users to search the database of restaurants
- In Tutorial 17, we will set up the initial version of `Patchy` and connect to Twitter's REST API via `HttpClient`

- In Tutorial 18, we will get rid of our HttpClient use and replace it with the JTwitter open source Java Twitter library
- In Tutorial 19, we will add in support for listening to the user's timeline and polling for updates via a local service
- In Tutorial 20, we will connect the client to the local service, displaying the timeline in a list
- In Tutorial 21, we will split the service into a separate application and connect to it as a remote service
- In Tutorial 22, we will start to monitor for status updates from certain people and will raise a `Notification` when such updates are received
- In Tutorial 23, we will give the users a menu option to embed their location into their own status update, to let people know where they are
- In Tutorial 24, we will watch for other status updates containing locations and will display them on a map
- In Tutorial 25, we add support for playing video clips, to serve as extensions to an online help system
- In Tutorial 26, we add support for displaying local HTML files, to form the foundations of an online help system
- In Tutorial 27, we augment the local HTML with dynamic data pulled from the user preferences, to round out the help system
- In Tutorial 28, we add a button to request manual updates to the timeline and make some changes to the list selector in the timeline list
- In Tutorial 29, we create our own custom widget for entering status updates, and then use that widget in the main user interface
- In Tutorial 30, we add a menu choice to hide or show the status update widget, and we make it appear and disappear in an animated fashion
- In Tutorial 31, we change our timeline list selector again and change the background of a button
- In Tutorial 32, we download and add a custom font to our application and use it in the status entry widget

- In Tutorial 33, we add preliminary support to `LunchList` to take photos of a restaurant using the built-in camera
- In Tutorial 34, we allow you to randomly choose a restaurant for lunch by shaking the device
- In Tutorial 35, we switch `Patchy` from using a callback method to get timeline updates to using a broadcast `Intent`
- In Tutorial 36, we create a fresh (albeit disposable) project to demonstrate introspection techniques, to help you discover third-party application capabilities at runtime
- In Tutorial 37, we tie Twitter friends to the Android contacts database
- In Tutorial 38, we expose our restaurant list in `LunchList` as a content provider
- In Tutorial 39, we further tie Twitter friends to the Android contacts database, highlighting them in the list of friends
- In Tutorial 40, we monitor the battery state and poll Twitter less frequently when the battery gets low
- In Tutorial 41, we enable Twitter monitoring even when the device is asleep

PART I – Introductory Tutorials

TUTORIAL 1

Your First Android App

This tutorial will help you get your very first Android application up and running, using just the files generated by Android's development tools.

Step-By-Step Instructions

Here is how you can create this application:

Step #1: Choose a Place For Your Applications

You will need a spot on your development machine to store all of the projects that you will create via the tutorials in this book.

Linux

If you are running Linux, open up a terminal window (e.g., Applications > Accessories > Terminal in Ubuntu 8.10) and execute the following commands:

```
cd ~
mkdir AndroidTutorials
cd AndroidTutorials
```

This will create an `AndroidTutorials` directory in your home directory and move you into it.

Windows XP

If you are running Windows XP, do the following:

1. Open up `My Documents`
2. Create a new folder named `AndroidTutorials` in `My Documents`

Step #2: Check Your Java Environment

You need to have Sun's Java SE version 1.5 or 1.6 in order to compile Android applications.

Linux

Using the terminal, execute the following command:

```
javac -version
```

If the output includes a line akin to `javac 1.6.0_10`, your Java version should be fine. If you do not believe you have the proper Java version, you should install the correct one, either through your Linux distribution or through Sun's Java site.

Windows XP

Open a Command Prompt by clicking on the **Start** menu, choosing **Run...**, entering `cmd` in the **Open:** field, and clicking **OK**. Then, execute the following command:

```
javac -version
```

You should see output akin to:

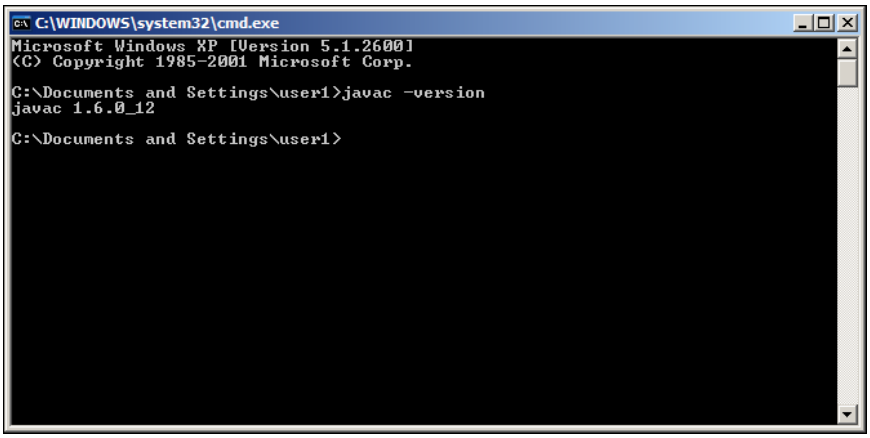


Figure 1. Output from Java compiler version test

If you do not, you will need to install the Java SE SDK from Sun's Java site. You should also be sure to add the directory for the Java commands to your PATH by:

1. Finding where the Java commands are (e.g., `C:\Program Files\Java\jdk1.6.0_12\bin`).
2. Go to your Control Panel (**Start** > **Settings** > **Control Panel**).
3. Double-click on the **System** applet.
4. Click the **Advanced** tab.
5. Click the **Environment Variables** button.
6. If there is a PATH value in the **User variables** area at the top, add your path to the end by double-clicking the existing one, scrolling to the end, typing a semicolon (;) and the path from step #1 above. If there is no such PATH value, click the **New** button, fill in PATH as the **Variable name** and the path from step #1 above as the **Variable value**.

Step #3: Download and Install the Android SDK

The Android 1.1 SDK is available from http://developer.android.com – just follow the download link and choose the ZIP file appropriate for your platform.

Linux

Once you have downloaded the SDK, you will need to unZIP it somewhere appropriate on your PC. Some people prefer to have such applications set up in their home directory; others prefer programs reside elsewhere (e.g., `/opt`). Choose a location and unZIP the archive there. This will create an `android-sdk-linux_x86-1.1_r1` directory. Remember this location, as this and future tutorials will refer to this location as "where you installed the Android SDK".

Windows XP

Once you have downloaded the SDK, you will need to unZIP it somewhere appropriate on your PC. Note that there is no program installer, so unZIPping the archive is all you need to do. You can unZIP it to your desktop, to a fake company in `C:\Program Files`, in the root of `C:\`, or wherever. When you unZIP it, it should create a directory named `android-sdk-windows-1.1_r1`. Remember this location, as this and future tutorials will refer to this location as "where you installed the Android SDK".

Optional for All Platforms

Per the Android documentation: "Optionally, you can add the path to the SDK tools directory to your path. The `tools/` directory is located in the SDK directory.

- On Linux, edit your `~/.bash_profile` or `~/.bashrc` file. Look for a line that sets the `PATH` environment variable and add the full path to the `tools/` directory to it. If you don't see a line setting the path, you can add one: `export PATH = ${PATH}:<your_sdk_dir>/tools`
- On a Mac, look in your home directory for `.bash_profile` and proceed as for Linux. You can create the `.bash_profile`, if you haven't already set one up on your machine.
- On Windows, right click on My Computer, and select Properties. Under the Advanced tab, hit the Environment Variables button, and in the dialog that comes up, double-click on Path under System Variables. Add the full path to the `tools/` directory to the path."

Step #4: Generate the Application Files

Inside your terminal (e.g., Command Prompt for Windows), move back into the `AndroidTutorials` directory you created in step #1. Then, run the following command:

```
activitycreator --out FirstApp apt.tutorial.FirstApp
```

This will create an application skeleton for you, complete with everything you need to build your first Android application: Java source code, build instructions, etc.

Step #5: Examine and Modify the Layout File

Using your favorite text editor, look at `FirstApp/res/layout/main.xml`. It should look like this:

```
<?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"
    >
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Hello Mark"
    />
</LinearLayout>
```

You will see an XML element for a `LinearLayout` wrapping an XML element for a `TextView`. Inside the `TextView`, you will see an attribute named `android:text` with a value of `Hello World, FirstApp`. Change that value to `Hello,` plus your name (e.g., `Hello, Mark`). Save your changes to this file.

Step #6: Examine the Activity Java Source

Next, take a look at `FirstApp/src/apt/tutorial/FirstApp.java`. It should look like this:

```
package apt.tutorial;

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

public class FirstApp extends Activity
{
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

Notice how the activity class does not have a constructor, only an `onCreate()` method. Also notice that the `onCreate()` method calls `setContentView(R.layout.main)`, which is how Android knows to load the layout you saw in step #5 and display it on the screen.

You do not need to make any changes to this file for this tutorial.

Step #7: Install Ant

To build applications, you will need the Apache Ant build tool. This can be downloaded from the Ant Web site, and instructions for installing it can be found at the Ant online manual.

Linux users: You may be able to install Ant through your Linux distribution's package management system (e.g., Synaptic in Ubuntu). However, that version of Ant may be set up to use the GNU `gcj` Java environment rather than Sun's, which may cause you problems. Make sure that however you set up Ant, it uses the Sun JDK you installed in step #2

Step #8: Compile the Application

In your terminal, change into the `FirstApp` directory, then run the following command:

```
ant
```

This will compile the application. It should emit a list of steps involved in the installation process, which look like this:

```
Buildfile: build.xml

dirs:
 [echo] Creating output directories if needed...
 [mkdir] Created dir: /home/mark/AndroidTutorials/FirstApp/bin/classes

resource-src:
 [echo] Generating R.java / Manifest.java from the resources...

aidl:
 [echo] Compiling aidl files into Java classes...

compile:
 [javac] Compiling 2 source files to
/home/mark/AndroidTutorials/FirstApp/bin/classes

dex:
 [echo] Converting compiled files and external libraries into bin/classes.dex...

package-res:

package-res-no-assets:
 [echo] Packaging resources...

debug:
 [echo] Packaging bin/FirstApp-debug.apk, and signing it with a debug key...
 [exec] Using keystore: /home/mark/.android/debug.keystore

BUILD SUCCESSFUL
Total time: 1 second
```

Note the `BUILD SUCCESSFUL` at the bottom – that is how you know the application compiled successfully.

Step #9: Start Your Emulator

Android developers typically use the Android emulator to test and debug their applications. This is a copy of the Android firmware set up to run in a virtual machine on your development workstation. Note that running the emulator for the first time will take longer than subsequent runs, and you

can usually just keep the emulator up and running – you do not need to stop and restart it after every rebuild of your application.

The emulator is located in the `tools/` directory of your Android SDK installation.

Linux

If you added `tools/` to your `PATH` in step #3, just run the following command:

```
emulator &
```

If you did not add `tools/` to your `PATH`, fully-qualify the path to emulator in your Android SDK `tools/` directory.

Windows XP

Create a desktop icon for the emulator by:

1. Opening up the My Computer window
2. Navigating to the `tools/` directory of your Android SDK installation
3. While holding down the <Alt> key, drag `emulator.exe` from the `tools/` directory to your desktop, which will create a shortcut icon to that program

Then, launch the emulator using that icon.

The emulator should look something like this:

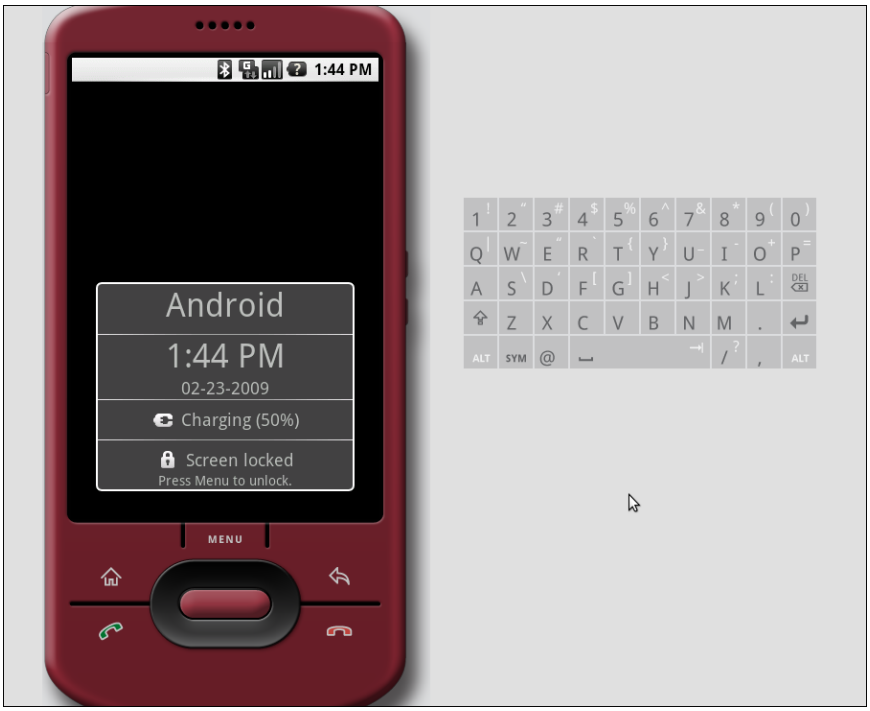


Figure 2. Android emulator

Step #10: Install the Application in Your Emulator

Back in your terminal window, run the following command:

```
ant install
```

This will both compile and install your application in the emulator.

NOTE: The second and subsequent times you install the same application, you will need to use the following command instead:

```
ant reinstall
```

Step #11: Run the Application in Your Emulator

Click the [MENU] button in the emulator window to bring up the Android home screen:

Figure 3. Android emulator home screen

Then, click on the grey button at the bottom of the emulator screen, just above the [MENU] button, to open up the application launcher:

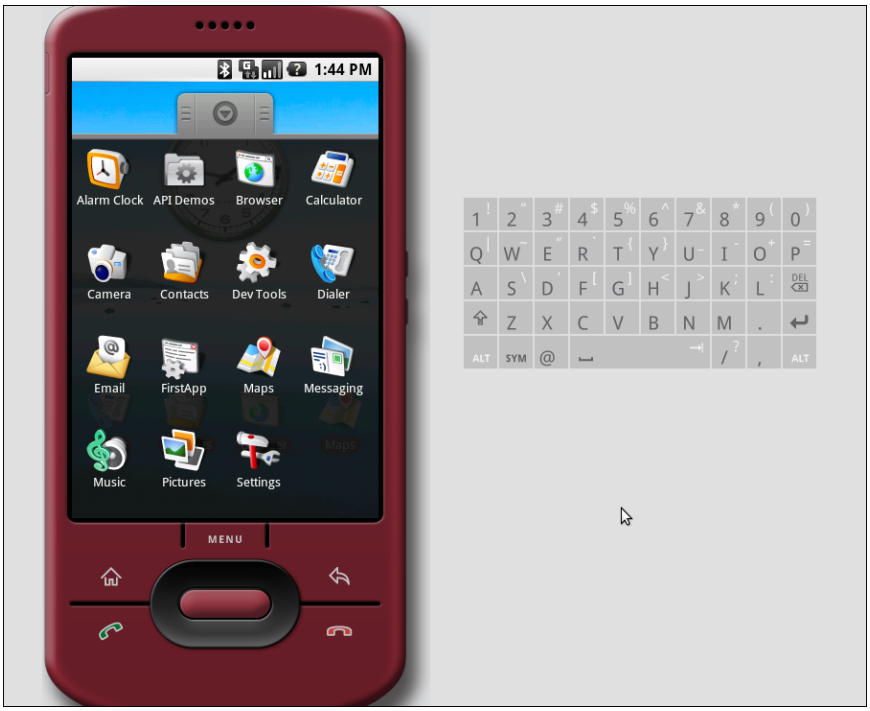


Figure 4. Android emulator application launcher

Notice there is an icon for your `FirstApp` application. Click on it to open it and see your first activity in action:

Figure 5. Your FirstApp

To leave the application and return to the launcher, press the "back button", located below and to the right of the [MENU] button, and looks like an arrow pointing to the left.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Instead of using the console tools as documented above, try using Eclipse. You will need to download Eclipse, install the Android plug-in, and use it to create your first project.
- If you have an Android device, try installing the app on the device and running it there. The easiest way to do this is to shut down your emulator, plug in your device, and run `ant install` again.

- Play around with the values for `android:layout_width` and `android:layout_height`. You might also add `android:background = "#FFFF0000"` to the `FirstApp/res/layout/main.xml` file, to give the screen a red background, so you can see where the widget ends and the rest of the screen begins.

A Simple Form

This tutorial is the first of several that will build up a "lunch list" application, where you can track various likely places to go to lunch. While this application may seem silly, it will give you a chance to exercise many features of the Android platform. Besides, perhaps you may even find the application to be useful someday.

Step-By-Step Instructions

Here is how you can create this application:

Step #1: Generate the Application Skeleton

Inside your terminal (e.g., Command Prompt for Windows), move back into the `AndroidTutorials` directory you created in step #1 of the first tutorial. Then, run the following command:

```
activitycreator --out LunchList apt.tutorial.LunchList
```

This will create an application skeleton for you, complete with everything you need to start building the Lunch List application.

Step #2: Modify the Layout

Using your text editor, open the `LunchList/res/layout/main.xml` file. Initially, that file will look like this:

```
<?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"
 >
<TextView
 android:layout_width="fill_parent"
 android:layout_height="wrap_content"
 android:text="Hello World, LunchList"
 />
</LinearLayout>
```

Change that layout to look like this:

```
<?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"
  >
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    >
    <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Name:"
      />
    <EditText android:id="@+id/name"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      />
  </LinearLayout>
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    >
    <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Address:"
```

```
      />
    <EditText android:id="@+id/addr"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      />
  </LinearLayout>
  <Button android:id="@+id/save"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Save"
  />
</LinearLayout>
```

This gives us a three-row form: one row with a labeled field for the restaurant name, one with a labeled field for the restaurant address, and a big Save button.

Step #3: Compile and Install the Application

Compile and install the application in the emulator by running the following command in your terminal:

```
ant reinstall
```

Since both this application and the one from the previous tutorial share the `apt.tutorial` namespace, only one of the two applications can be installed at one time. That is why we need to "reinstall" this application on this compile step.

Step #4: Run the Application in the Emulator

In your emulator, in the application launcher, you will see an icon for your `LunchList` application. Click it to bring up your form:

Figure 6. The first edition of LunchList

Use the directional pad (D-pad) below the [MENU] button to navigate between the fields and button. Enter some text in the fields and click the button, to see how those widgets behave. Then, click the back button to return to the application launcher.

Step #5: Create a Model Class

Now, we want to add a class to the project that will hold onto individual restaurants that will appear in the `LunchList`. Right now, we can only really work with one restaurant, but that will change in a future tutorial.

So, using your text editor, create a new file named `LunchList/src/apt/tutorial/Restaurant.java` with the following contents:

```
package apt.tutorial;
```

```
public class Restaurant {
  private String name="";
  private String address="";

  public String getName() {
    return(name);
  }

  public void setName(String name) {
    this.name=name;
  }

  public String getAddress() {
    return(address);
  }

  public void setAddress(String address) {
    this.address=address;
  }
}
```

This is simply a rudimentary model, with private data members for the name and address, and getters and setters for each of those.

Of course, don't forget to save your changes!

Step #6: Save the Form to the Model

Finally, we want to hook up the Save button, such that when it is pressed, we update a `Restaurant` object based on the two `EditText` fields. To do this, open up the `LunchList/src/apt/tutorial/LunchList.java` file and replace the generated `Activity` implementation with the one shown below:

```
package apt.tutorial;

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

public class LunchList extends Activity {
  Restaurant r=new Restaurant();

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

```
    setContentView(R.layout.main);

    Button save=(Button)findViewById(R.id.save);

    save.setOnClickListener(onSave);
  }

  private View.OnClickListener onSave=new View.OnClickListener() {
    public void onClick(View v) {
      EditText name=(EditText)findViewById(R.id.name);
      EditText address=(EditText)findViewById(R.id.addr);

      r.setName(name.getText().toString());
      r.setAddress(address.getText().toString());
    }
  };
}
```

Here, we:

- Create a single local `Restaurant` instance when the activity is instantiated
- Get our `Button` from the `Activity` via `findViewById()`, then connect it to a listener to be notified when the button is clicked
- In the listener, we get our two `EditText` widgets via `findViewById()`, then retrieve their contents and put them in the `Restaurant`

Then, re-run the `ant reinstall` command to compile and update the emulator. Run the application to make sure it seems like it runs without errors, though at this point we are not really using the data saved in the `Restaurant` object just yet.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Instead of using the console tools as documented above, try using Eclipse. You will need to download Eclipse, install the Android plug-in, and use it to create your first project.
- Try replacing the icon for your application. To do this, you will need to find a suitable 48x48 pixel image, create a `drawable/` directory

inside your `res/` directory in the project, and adjust the `AndroidManifest.xml` file to contain an `android:icon = "@drawable/my_icon"` attribute in the application element, where `my_icon` is replaced by the base name of your image file.

- Try playing with the fonts for use in both the `TextView` and `EditText` widgets. The Android SDK documentation will show a number of XML attributes you can manipulate to change the color, make the text boldface, etc.

A Fancier Form

In this tutorial, we will switch to using a `TableLayout` for our restaurant data entry form, plus add a set of radio buttons to represent the type of restaurant.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `02-SimpleForm` edition of `LunchList` to use as a starting point.

Step #1: Switch to a TableLayout

First, open `LunchList/res/layout/main.xml` and modify its contents to look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:stretchColumns="1"
  >
  <TableRow>
    <TextView android:text="Name:" />
    <EditText android:id="@+id/name" />
  </TableRow>
  <TableRow>
```

```
    <TextView android:text="Address:" />
    <EditText android:id="@+id/addr" />
  </TableRow>
  <Button android:id="@+id/save"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Save"
  />
</TableLayout>
```

Notice that we replaced the three `LinearLayout` containers with a `TableLayout` and two `TableRow` containers. We also set up the `EditText` column to be stretchable.

Recompile and reinstall the application, then run it in the emulator. You should see something like this:

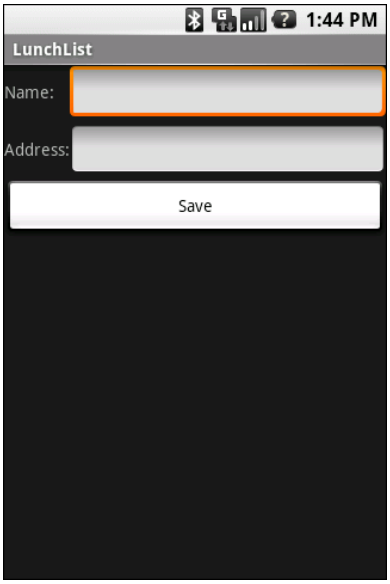


Figure 7. Using a TableLayout

Notice how the two `EditText` fields line up, whereas before, they appeared immediately after each label.

Step #2: Add a RadioGroup

Next, we should add some `RadioButton` widgets to indicate the type of restaurant this is: one that offers take-out, one where we can sit down, or one that is only a delivery service.

To do this, modify `LunchList/res/layout/main.xml` once again, this time to look like:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:stretchColumns="1"
  >
  <TableRow>
    <TextView android:text="Name:" />
    <EditText android:id="@+id/name" />
  </TableRow>
  <TableRow>
    <TextView android:text="Address:" />
    <EditText android:id="@+id/addr" />
  </TableRow>
  <TableRow>
    <TextView android:text="Type:" />
    <RadioGroup android:id="@+id/types">
      <RadioButton android:id="@+id/take_out"
        android:text="Take-Out"
      />
      <RadioButton android:id="@+id/sit_down"
        android:text="Sit-Down"
      />
      <RadioButton android:id="@+id/delivery"
        android:text="Delivery"
      />
    </RadioGroup>
  </TableRow>
  <Button android:id="@+id/save"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Save"
  />
</TableLayout>
```

Our `RadioGroup` and `RadioButton` widgets go inside the `TableLayout`, so they will line up with the rest of table – you can see this once you recompile, reinstall, and run the application:

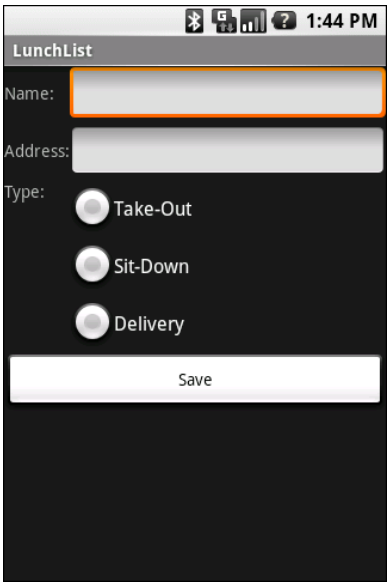


Figure 8. Adding radio buttons

Note that there is no pre-selected restaurant type. There is no good way in the layout to indicate which is the default selection (if any). Instead, you can handle that in Java code, by calling `toggle()` on the `RadioButton` to be made the default as part of your `onCreate()` method.

Step #3: Update the Model

Right now, our model class has no place to hold the restaurant type. To change that, modify `LunchList/src/apt/tutorial/Restaurant.java` to add in a new `private String type` data member and a getter/setter pair, like these:

```
public String getType() {
  return(type);
}

public void setType(String type) {
  this.type=type;
}
```

When you are done, your `Restaurant` class should look something like this:

```
package apt.tutorial;

public class Restaurant {
  private String name="";
  private String address="";
  private String type="";

  public String getName() {
    return(name);
  }

  public void setName(String name) {
    this.name=name;
  }

  public String getAddress() {
    return(address);
  }

  public void setAddress(String address) {
    this.address=address;
  }

  public String getType() {
    return(type);
  }

  public void setType(String type) {
    this.type=type;
  }
}
```

Step #4: Save the Type to the Model

Finally, we need to wire our `RadioButton` widgets to the model, such that when the user clicks the Save button, the type is saved as well. To do this, modify the `onSave` listener object to look like this:

```
private View.OnClickListener onSave=new View.OnClickListener() {
  public void onClick(View v) {
    EditText name=(EditText)findViewById(R.id.name);
    EditText address=(EditText)findViewById(R.id.addr);

    r.setName(name.getText().toString());
    r.setAddress(address.getText().toString());

    RadioGroup types=(RadioGroup)findViewById(R.id.types);

    switch (types.getCheckedRadioButtonId()) {
      case R.id.sit_down:
```

```
        r.setType("sit_down");
        break;

      case R.id.take_out:
        r.setType("take_out");
        break;

      case R.id.delivery:
        r.setType("delivery");
        break;
    }
  }
};
```

Note that you will also need to import `android.widget.RadioGroup` for this to compile. The full activity will then look like this:

```
package apt.tutorial;

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

public class LunchList extends Activity {
  Restaurant r=new Restaurant();

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

    Button save=(Button)findViewById(R.id.save);

    save.setOnClickListener(onSave);
  }

  private View.OnClickListener onSave=new View.OnClickListener() {
    public void onClick(View v) {
      EditText name=(EditText)findViewById(R.id.name);
      EditText address=(EditText)findViewById(R.id.addr);

      r.setName(name.getText().toString());
      r.setAddress(address.getText().toString());

      RadioGroup types=(RadioGroup)findViewById(R.id.types);

      switch (types.getCheckedRadioButtonId()) {
        case R.id.sit_down:
          r.setType("sit_down");
```

```
          break;

        case R.id.take_out:
          r.setType("take_out");
          break;

        case R.id.delivery:
          r.setType("delivery");
          break;
      }
    }
  };
}
```

Recompile, reinstall, and run the application. Confirm that you can save the restaurant data without errors.

If you are wondering what will happen if there is no selected `RadioButton`, the `RadioGroup` call to `getCheckedRadioButtonId()` will return `-1`, which will not match anything in our `switch` statement, and so the model will not be modified.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- If you have an Android device, try installing the app on the device and running it there. The easiest way to do this is to shut down your emulator, plug in your device, and run `ant reinstall`.
- Set one of the three radio buttons to be selected by default, as described in step #2 above.
- Try creating the `RadioButton` widgets in Java code, instead of in the layout. To do this, you will need to create the `RadioButton` objects themselves, configure them (e.g., supply them with text to display), then add them to the `RadioGroup` via `addView()`.
- Try adding more `RadioButton` widgets than there are room to display on the screen. Note how the screen does not automatically scroll to show them. Then, wrap your entire layout in a `ScrollView` container, and see how the form can now scroll to accommodate all of your widgets.

Adding a List

In this tutorial, we will change our model to be a list of `Restaurant` objects, rather than just one. Then, we will add a `ListView` to view the available `Restaurant` objects. This will be rather incomplete, in that we can only add a new `Restaurant`, not edit or delete an existing one, but we will cover those steps too in a later tutorial.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `03-FancierForm` edition of `LunchList` to use as a starting point.

Step #1: Hold a List of Restaurants

First, if we are going to have a list of `Restaurant` objects in the UI, we need a list of `Restaurant` objects as our model. So, in `LunchList`, change:

```
Restaurant r=new Restaurant();
```

to:

```
List<Restaurant> model=new ArrayList<Restaurant>();
```

Note that you will need to import `java.util.List` and `java.util.ArrayList` as well.

Step #2: Save Adds to List

Note that the above code will not compile, because our `onSave` `Button` click handler is still set up to reference the old single `Restaurant` model. For the time being, we will have `onSave` simply add a new `Restaurant`.

All we need to do is add a local `Restaurant` `r` variable, populate it, and add it to the list:

```
private View.OnClickListener onSave=new View.OnClickListener() {
  public void onClick(View v) {
    Restaurant r=new Restaurant();
    EditText name=(EditText)findViewById(R.id.name);
    EditText address=(EditText)findViewById(R.id.addr);

    r.setName(name.getText().toString());
    r.setAddress(address.getText().toString());

    RadioGroup types=(RadioGroup)findViewById(R.id.types);

    switch (types.getCheckedRadioButtonId()) {
      case R.id.sit_down:
        r.setType("sit_down");
        break;

      case R.id.take_out:
        r.setType("take_out");
        break;

      case R.id.delivery:
        r.setType("delivery");
        break;
    }
  }
};
```

At this point, you should be able to rebuild and reinstall the application. Test it out to make sure that clicking the button does not cause any unexpected errors.

Step #3: Implement toString()

To simplify the creation of our `ListView`, we need to have our `Restaurant` class respond intelligently to `toString()`. That will be called on each `Restaurant` as it is displayed in our list.

For the purposes of this tutorial, we will simply use the name – later tutorials will make the rows much more interesting and complex.

So, add a `toString()` implementation on `Restaurant` like this:

```
public String toString() {
  return(getName());
}
```

Recompile and ensure your application still builds.

Step #4: Add a ListView Widget

Now comes the challenging part – adding the `ListView` to the layout.

The challenge is in getting the layout right. Right now, while we have only the one screen to work with, we need to somehow squeeze in the list without eliminating space for anything else. In fact, ideally, the list takes up all the available space that is not being used by our current details form.

One way to achieve that is to use a `RelativeLayout` as the over-arching layout for the screen. We anchor the details form to the bottom of the screen, then have the list span the space from the top of the screen to the top of the details form.

To make this change, replace your current `LunchList/res/layout/main.xml` with the following:

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

```
  android:layout_height="fill_parent"
  >
  <TableLayout android:id="@+id/details"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:stretchColumns="1"
    >
    <TableRow>
      <TextView android:text="Name:" />
      <EditText android:id="@+id/name" />
    </TableRow>
    <TableRow>
      <TextView android:text="Address:" />
      <EditText android:id="@+id/addr" />
    </TableRow>
    <TableRow>
      <TextView android:text="Type:" />
      <RadioGroup android:id="@+id/types">
        <RadioButton android:id="@+id/take_out"
          android:text="Take-Out"
        />
        <RadioButton android:id="@+id/sit_down"
          android:text="Sit-Down"
        />
        <RadioButton android:id="@+id/delivery"
          android:text="Delivery"
        />
      </RadioGroup>
    </TableRow>
    <Button android:id="@+id/save"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="Save"
    />
  </TableLayout>
  <ListView android:id="@+id/restaurants"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentTop="true"
    android:layout_above="@id/details"
  />
</RelativeLayout>
```

If you recompile and rebuild the application, then run it, you will see our form slid to the bottom, with empty space at the top:

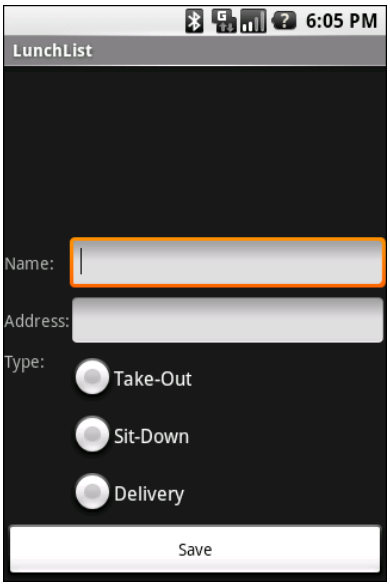


Figure 9. Adding a list to the top and sliding the form to the bottom

Step #5: Build and Attach the Adapter

The `ListView` will remain empty, of course, until we do something to populate it. What we want is for the list to show our running lineup of `Restaurant` objects.

Since we have our `ArrayList<Restaurant>`, we can easily wrap it in an `ArrayAdapter<Restaurant>`. This also means, though, that when we add a `Restaurant`, we need to add it to the `ArrayAdapter` via `add()` – the adapter will, in turn, put it in the `ArrayList`. Otherwise, if we add it straight to the `ArrayList`, the adapter will not know about the added `Restaurant` and therefore will not display it.

Here is the new implementation of the `LunchList` class:

```
package apt.tutorial;

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

```
import android.widget.Button;
import android.widget.EditText;
import android.widget.ListView;
import android.widget.RadioGroup;
import java.util.ArrayList;
import java.util.List;

public class LunchList extends Activity {
  List<Restaurant> model=new ArrayList<Restaurant>();
  ArrayAdapter<Restaurant> adapter=null;

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

    Button save=(Button)findViewById(R.id.save);

    save.setOnClickListener(onSave);

    ListView list=(ListView)findViewById(R.id.restaurants);

    adapter=new ArrayAdapter<Restaurant>(this,
                      android.R.layout.simple_list_item_1,
                      model);
    list.setAdapter(adapter);
  }

  private View.OnClickListener onSave=new View.OnClickListener() {
    public void onClick(View v) {
      Restaurant r=new Restaurant();
      EditText name=(EditText)findViewById(R.id.name);
      EditText address=(EditText)findViewById(R.id.addr);

      r.setName(name.getText().toString());
      r.setAddress(address.getText().toString());

      RadioGroup types=(RadioGroup)findViewById(R.id.types);

      switch (types.getCheckedRadioButtonId()) {
        case R.id.sit_down:
          r.setType("sit_down");
          break;

        case R.id.take_out:
          r.setType("take_out");
          break;

        case R.id.delivery:
          r.setType("delivery");
          break;
      }

      adapter.add(r);
```

```
    }
  };
}
```

The magic value `android.R.layout.simple_list_item_1` is a stock layout for a list row, just displaying the text of the object in white on a black background with a reasonably large font. In later tutorials, we will change the look of our rows to suit our own designs.

If you then add a few restaurants via the form, it will look something like this:

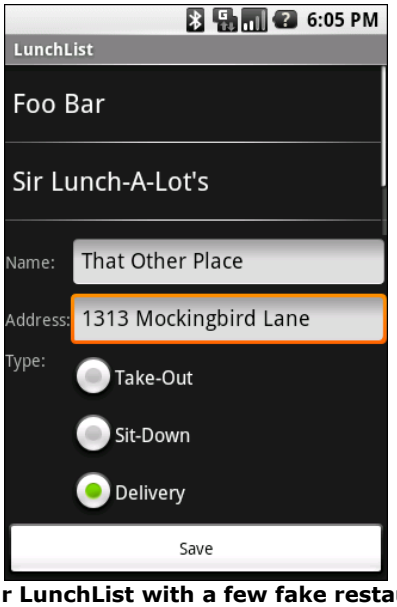


Figure 10. Our LunchList with a few fake restaurants added

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- See what the activity looks like if you use a `Spinner` instead of a `ListView`.
- Make the address field, presently an `EditText` widget, into an `AutoCompleteTextView`, using the other addresses as values to possibly

reuse (e.g., for multiple restaurants in one place, such as a food court or mall).

- Try designing your own layout and using it instead of the stock one in the `ListView`. The simplest way to do this is to make your own layout simply be a `TextView` (styled as you wish), then supply your layout resource ID in the `ArrayAdapter` constructor.

Making Our List Be Fancy

In this tutorial, we will update the layout of our `ListView` rows, so they show both the name and address of the `Restaurant`, plus an icon indicating the type. Along the way, we will need to create our own custom `ListAdapter` to handle our row views and a `RestaurantWrapper` to populate a row from a `Restaurant`.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `04-ListView` edition of `LunchList` to use as a starting point.

Step #1: Create a Stub Custom Adapter

First, let us create a stub implementation of a `RestaurantAdapter` that will be where we put our logic for creating our own custom rows. That can look like this, implemented as an inner class of `LunchList`:

```
class RestaurantAdapter extends ArrayAdapter<Restaurant> {
  RestaurantAdapter() {
    super(LunchList.this,
          android.R.layout.simple_list_item_1,
          model);
  }
}
```

We hard-wire in the `android.R.layout.simple_list_item_1` layout for now, and we get our `Activity` and model from `LunchList` itself.

We also need to change our data member to be a `RestaurantAdapter`, both where it is declared and where it is instantiated in `onCreate()`. Make these changes, then rebuild and reinstall the application and confirm it works as it did at the end of the previous tutorial.

Step #2: Design Our Row

Next, we want to design a row that incorporates all three of our model elements: name, address, and type. For the type, we will use three icons, one for each specific type (sit down, drive-through, delivery). You can use whatever icons you wish, or you can get the icons used in this tutorial from the tutorial ZIP file that you can download.

The general layout is to have the icon on the left and the name stacked atop the address to the right:

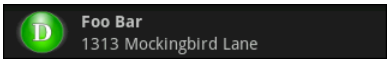


Figure 11. A fancy row for our fancy list

To achieve this look, we use a nested pair of `LinearLayout` containers. Use the following XML as the basis for `LunchList/res/layout/row.xml`:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:orientation="horizontal"
  android:padding="4px"
  >
  <ImageView android:id="@+id/icon"
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
    android:layout_alignParentTop="true"
    android:layout_alignParentBottom="true"
    android:layout_marginRight="4px"
  />
  <LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
```

```
    android:orientation="vertical"
    >
    <TextView android:id="@+id/name"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:gravity="center_vertical"
      android:textStyle="bold"
      android:singleLine="true"
      android:ellipsize="end"
    />
    <TextView android:id="@+id/address"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:gravity="center_vertical"
      android:singleLine="true"
      android:ellipsize="end"
    />
  </LinearLayout>
</LinearLayout>
```

Some of the unusual attributes applied in this layout include:

- `android:padding`, which arranges for some whitespace to be put outside the actual widget contents but still be considered part of the widget (or container) itself when calculating its size
- `android:textStyle`, where we can indicate that some text is in bold or italics
- `android:singleLine`, which, if `true`, indicates that text should not word-wrap if it extends past one line
- `android:ellipsize`, which indicates where text should be truncated and ellipsized if it is too long for the available space

Step #3: Override getView(): The Simple Way

Next, we need to use this layout ourselves in our `RestaurantAdapter`. To do this, we need to override `getView()` and inflate the layout as needed for rows.

Modify `RestaurantAdapter` to look like the following:

```
class RestaurantAdapter extends ArrayAdapter<Restaurant> {
  RestaurantAdapter() {
```

```
    super(LunchList.this,
          android.R.layout.simple_list_item_1,
          model);
  }

  public View getView(int position, View convertView,
                      ViewGroup parent) {
    View row=convertView;

    if (row==null) {
      LayoutInflater inflater=getLayoutInflater();

      row=inflater.inflate(R.layout.row, null);
    }

    Restaurant r=model.get(position);

    ((TextView)row.findViewById(R.id.name)).setText(r.getName());
    ((TextView)row.findViewById(R.id.address)).setText(r.getAddress());

    ImageView icon=(ImageView)row.findViewById(R.id.icon);

    if (r.getType().equals("sit_down")) {
      icon.setImageResource(R.drawable.ball_red);
    }
    else if (r.getType().equals("take_out")) {
      icon.setImageResource(R.drawable.ball_yellow);
    }
    else {
      icon.setImageResource(R.drawable.ball_green);
    }

    return(row);
  }
}
```

Notice how we create a row only if needed, recycling existing rows. But, we still pick out each `TextView` and `ImageView` from each row and populate it from the `Restaurant` at the indicated position.

Step #4: Create a RestaurantWrapper

To improve performance and encapsulation, we should move the logic that populates a row from a `Restaurant` into a separate class, one that can cache the `TextView` and `ImageView` widgets.

To do this, add the following inner class to `LunchList`:

```
class RestaurantWrapper {
  private TextView name=null;
  private TextView address=null;
  private ImageView icon=null;
  private View row=null;

  RestaurantWrapper(View row) {
    this.row=row;
  }

  void populateFrom(Restaurant r) {
    getName().setText(r.getName());
    getAddress().setText(r.getAddress());

    if (r.getType().equals("sit_down")) {
      getIcon().setImageResource(R.drawable.ball_red);
    }
    else if (r.getType().equals("take_out")) {
      getIcon().setImageResource(R.drawable.ball_yellow);
    }
    else {
      getIcon().setImageResource(R.drawable.ball_green);
    }
  }

  TextView getName() {
    if (name==null) {
      name=(TextView)row.findViewById(R.id.name);
    }

    return(name);
  }

  TextView getAddress() {
    if (address==null) {
      address=(TextView)row.findViewById(R.id.address);
    }

    return(address);
  }

  ImageView getIcon() {
    if (icon==null) {
      icon=(ImageView)row.findViewById(R.id.icon);
    }

    return(icon);
  }
}
```

Step #5: Recycle Rows via RestaurantWrapper

To take advantage of the new `RestaurantWrapper`, we need to modify `getView()` in `RestaurantAdapter`. Following the holder pattern, we need to create a `RestaurantWrapper` when we inflate a new row, cache that wrapper in the row via `setTag()`, then get it back later via `getTag()`.

Change `getView()` to look like the following:

```
public View getView(int position, View convertView,
                    ViewGroup parent) {
  View row=convertView;
  RestaurantWrapper wrapper=null;

  if (row==null) {
    LayoutInflater inflater=getLayoutInflater();

    row=inflater.inflate(R.layout.row, null);
    wrapper=new RestaurantWrapper(row);
    row.setTag(wrapper);
  }
  else {
    wrapper=(RestaurantWrapper)row.getTag();
  }

  wrapper.populateFrom(model.get(position));

  return(row);
}
```

This means the whole `LunchList` class looks like:

```
package apt.tutorial;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.view.LayoutInflater;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ImageView;
import android.widget.ListView;
import android.widget.RadioGroup;
import android.widget.TextView;
import java.util.ArrayList;
import java.util.List;
```

```
public class LunchList extends Activity {
  List<Restaurant> model=new ArrayList<Restaurant>();
  RestaurantAdapter adapter=null;

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

    Button save=(Button)findViewById(R.id.save);

    save.setOnClickListener(onSave);

    ListView list=(ListView)findViewById(R.id.restaurants);

    adapter=new RestaurantAdapter();
    list.setAdapter(adapter);
  }

  private View.OnClickListener onSave=new View.OnClickListener() {
    public void onClick(View v) {
      Restaurant r=new Restaurant();
      EditText name=(EditText)findViewById(R.id.name);
      EditText address=(EditText)findViewById(R.id.addr);

      r.setName(name.getText().toString());
      r.setAddress(address.getText().toString());

      RadioGroup types=(RadioGroup)findViewById(R.id.types);

      switch (types.getCheckedRadioButtonId()) {
        case R.id.sit_down:
          r.setType("sit_down");
          break;

        case R.id.take_out:
          r.setType("take_out");
          break;

        case R.id.delivery:
          r.setType("delivery");
          break;
      }

      adapter.add(r);
    }
  };

  class RestaurantAdapter extends ArrayAdapter<Restaurant> {
    RestaurantAdapter() {
      super(LunchList.this,
            android.R.layout.simple_list_item_1,
            model);
```

```
  }

  public View getView(int position, View convertView,
                      ViewGroup parent) {
    View row=convertView;
    RestaurantWrapper wrapper=null;

    if (row==null) {
      LayoutInflater inflater=getLayoutInflater();

      row=inflater.inflate(R.layout.row, null);
      wrapper=new RestaurantWrapper(row);
      row.setTag(wrapper);
    }
    else {
      wrapper=(RestaurantWrapper)row.getTag();
    }

    wrapper.populateFrom(model.get(position));

    return(row);
  }
}

class RestaurantWrapper {
  private TextView name=null;
  private TextView address=null;
  private ImageView icon=null;
  private View row=null;

  RestaurantWrapper(View row) {
    this.row=row;
  }

  void populateFrom(Restaurant r) {
    getName().setText(r.getName());
    getAddress().setText(r.getAddress());

    if (r.getType().equals("sit_down")) {
      getIcon().setImageResource(R.drawable.ball_red);
    }
    else if (r.getType().equals("take_out")) {
      getIcon().setImageResource(R.drawable.ball_yellow);
    }
    else {
      getIcon().setImageResource(R.drawable.ball_green);
    }
  }

  TextView getName() {
    if (name==null) {
      name=(TextView)row.findViewById(R.id.name);
    }
```

```
      return(name);
    }

    TextView getAddress() {
      if (address==null) {
        address=(TextView)row.findViewById(R.id.address);
      }

      return(address);
    }

    ImageView getIcon() {
      if (icon==null) {
        icon=(ImageView)row.findViewById(R.id.icon);
      }

      return(icon);
    }
  }
}
```

Rebuild and reinstall the application, then try adding several restaurants and confirm that, when the list is scrolled, everything appears as it should – the name, address, and icon all change.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Customize the rows beyond just the icon based on each `Restaurant`, such as applying different colors to the name based upon certain criteria.
- Use three different layouts for the three different `Restaurant` types. To do this, you will need to override `getItemViewType()` and `getViewTypeCount()` in the custom adapter to return the appropriate data.

Splitting the Tab

In this tutorial, we will move our `ListView` onto one tab and our form onto a separate tab of a `TabView`. Along the way, we will also arrange to update our form based on a `ListView` selections or clicks, even though the Save button will still only add new `Restaurant` objects to our list.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `05-FancyList` edition of `LunchList` to use as a starting point.

Step #1: Rework the Layout

First, we need to change our layout around, to introduce the tabs and split our UI between a list tab and a details tab. This involves:

- Removing the `RelativeLayout` and the layout attributes leveraging it, as that was how we had the list and form on a single screen
- Add in a `TabHost`, `TabWidget`, and `FrameLayout`, the latter of which is parent to the list and details

To accomplish this, replace your current `LunchList/res/layout/main.xml` with the following:

```
<?xml version="1.0" encoding="utf-8"?>
<TabHost xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@android:id/tabhost"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <TabWidget android:id="@android:id/tabs"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
  />
  <FrameLayout android:id="@android:id/tabcontent"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:paddingTop="62px"
  >
    <ListView android:id="@+id/restaurants"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
    />
    <TableLayout android:id="@+id/details"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:stretchColumns="1"
      android:paddingTop="4px"
      >
      <TableRow>
        <TextView android:text="Name:" />
        <EditText android:id="@+id/name" />
      </TableRow>
      <TableRow>
        <TextView android:text="Address:" />
        <EditText android:id="@+id/addr" />
      </TableRow>
      <TableRow>
        <TextView android:text="Type:" />
        <RadioGroup android:id="@+id/types">
          <RadioButton android:id="@+id/take_out"
            android:text="Take-Out"
          />
          <RadioButton android:id="@+id/sit_down"
            android:text="Sit-Down"
          />
          <RadioButton android:id="@+id/delivery"
            android:text="Delivery"
          />
        </RadioGroup>
      </TableRow>
      <Button android:id="@+id/save"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Save"
      />
    </TableLayout>
  </FrameLayout>
</TabHost>
```

Step #2: Wire In the Tabs

Next, we need to modify the `LunchList` itself, so it is a `TabActivity` (rather than a plain `Activity`) and teaches the `TabHost` how to use our `FrameLayout` contents for the individual tab panes. To do this:

1. Add imports to `LunchList` for `android.app.TabActivity` and `android.widget.TabHost`
2. Make `LunchList` extend `TabActivity`
3. Obtain 32px high icons from some source to use for the list and details tab icons, place them in `LunchList/res/drawable` as `list.png` and `restaurant.png`, respectively
4. Add the following code to the end of your `onCreate()` method:

```
TabHost.TabSpec spec=getTabHost().newTabSpec("tag1");

spec.setContent(R.id.restaurants);
spec.setIndicator("List", getResources()
                          .getDrawable(R.drawable.list));
getTabHost().addTab(spec);

spec=getTabHost().newTabSpec("tag2");
spec.setContent(R.id.details);
spec.setIndicator("Details", getResources()
                            .getDrawable(R.drawable.restaurant));
getTabHost().addTab(spec);

getTabHost().setCurrentTab(0);
```

At this point, you can recompile and reinstall the application and try it out. You should see a two-tab UI like this:

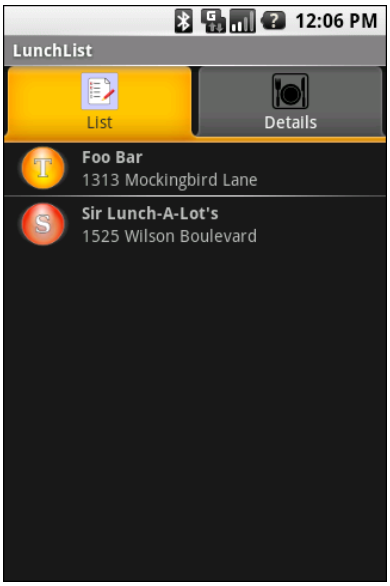


Figure 12. The first tab of the two-tab LunchList

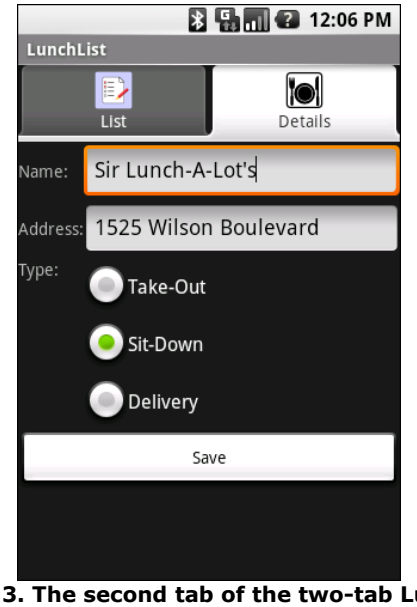


Figure 13. The second tab of the two-tab LunchList

Step #3: Get Control On List Events

Next, we need to detect when the user clicks on one of our `Restaurant` objects in the list, so we can update our detail form with that information.

First, add an import for `android.widget.AdapterView` to `LunchList`. Then, create an `AdapterView.OnItemClickListener` named `onListClick`:

```
private AdapterView.OnItemClickListener onListClick=new
AdapterView.OnItemClickListener() {
  public void onItemClick(AdapterView<?> parent,
                          View view, int position,
                          long id) {
  }
};
```

Finally, call `setOnItemClickListener()` on the `ListView` in the activity's `onCreate()` to connect the `ListView` to the `onListClick` listener object (`list.setOnItemClickListener(onListClick);`)

Step #4: Update Our Restaurant Form On Clicks

Next, now that we have control in a list item click, we need to actually find the associated `Restaurant` and update our details form.

To do this, you need to do two things. First, move the `name`, `address`, and `types` variables into data members and populate them in the activity's `onCreate()` – our current code has them as local variables in the `onSave` listener object's `onClick()` method. So, you should have some data members like:

```
EditText name=null;
EditText address=null;
RadioGroup types=null;
```

And some code after the call to `setContentView()` in `onCreate()` like:

```
name=(EditText)findViewById(R.id.name);
address=(EditText)findViewById(R.id.addr);
types=(RadioGroup)findViewById(R.id.types);
```

Then, add smarts to `onListClick` to update the details form:

```
private AdapterView.OnItemClickListener onListClick=new
AdapterView.OnItemClickListener() {
  public void onItemClick(AdapterView<?> parent,
                          View view, int position,
                          long id) {
    Restaurant r=model.get(position);

    name.setText(r.getName());
    address.setText(r.getAddress());

    if (r.getType().equals("sit_down")) {
      types.check(R.id.sit_down);
    }
    else if (r.getType().equals("take_out")) {
      types.check(R.id.take_out);
    }
    else {
      types.check(R.id.delivery);
    }
  }
};
```

Note how we find the clicked-upon `Restaurant` via the position parameter, which is an index into our `ArrayList` of `Restaurant` objects.

Step #5: Switch Tabs On Clicks

Finally, we want to switch to the details form when the user clicks a `Restaurant` in the list.

This is just one extra line of code, in the `onItemClick()` method of our `onListClick` listener object:

```
getTabHost().setCurrentTab(1);
```

This just changes the current tab to the one known as index `1`, which is the second tab (tabs start counting at `0`).

At this point, you should be able to recompile and reinstall the application and test out the new functionality.

Here is the complete source code to our `LunchList` activity, after all of the changes made in this tutorial.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Add a date in the `Restaurant` model to note the last time you visited the restaurant, then use either `DatePicker` or `DatePickerDialog` to allow users to set the date when they create their `Restaurant` objects.
- Try making a version of the activity that uses a `ViewFlipper` and a `Button` to flip from the list to the detail form, rather than using two tabs.

Menus and Messages

In this tutorial, we will add a `EditText` for a note to our details form and `Restaurant` model. Then, we will add an options menu that will display the note as a `Toast`.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `06-Tabs` edition of `LunchList` to use as a starting point.

Step #1: Add Notes to the Restaurant

First, our `Restaurant` model does not have any spot for notes. Add a `String notes` data member plus an associated getter and setter. Your resulting class should look like:

```
package apt.tutorial;

public class Restaurant {
  private String name="";
  private String address="";
  private String type="";
  private String notes="";

  public String getName() {
    return(name);
```

```
  }

  public void setName(String name) {
    this.name=name;
  }

  public String getAddress() {
    return(address);
  }

  public void setAddress(String address) {
    this.address=address;
  }

  public String getType() {
    return(type);
  }

  public void setType(String type) {
    this.type=type;
  }

  public String getNotes() {
    return(notes);
  }

  public void setNotes(String notes) {
    this.notes=notes;
  }

  public String toString() {
    return(getName());
  }
}
```

Step #2: Add Notes to the Detail Form

Next, we need `LunchList` to make use of the notes. To do this, first add the following `TableRow` above the Save button in our `TableLayout` in `LunchList/res/layout/main.xml`:

```
<TableRow>
  <TextView android:text="Notes:" />
  <EditText android:id="@+id/notes"
    android:singleLine="false"
    android:gravity="top"
    android:lines="2"
    android:scrollHorizontally="false"
    android:maxLines="2"
    android:maxWidth="200sp"
```

```
  />
</TableRow>
```

Then, we need to modify the `LunchList` activity itself, by:

1. Adding another data member for the `notes EditText` widget defined above
2. Find our `notes EditText` widget as part of `onCreate()`, like we do with other `EditText` widgets
3. Save our notes to our `Restaurant` in `onSave`
4. Restore our notes to the `EditText` in `onListClick`

At this point, you can recompile and reinstall the application to see your notes field in action:

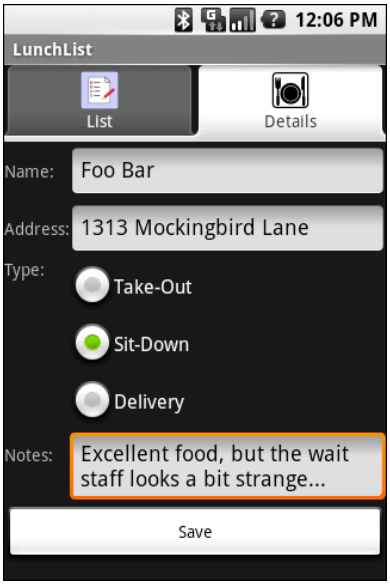


Figure 14. The notes field in the details form

Step #3: Define the Option Menu

Now, we need to create an option menu and arrange for it to be displayed when the user clicks the [MENU] button.

The menu itself can be defined as a small piece of XML. Enter the following as `LunchList/res/menu/option.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/toast"
    android:title="Raise Toast"
    android:icon="@drawable/toast"
  />
</menu>
```

This code relies upon an icon stored in `LunchList/res/drawable/toast.png`. Find something suitable to use, preferably around 32px high.

Then, to arrange for the menu to be displayed, add the following method to `LunchList`:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  new MenuInflater(getApplication())
                                .inflate(R.menu.option, menu);

  return(super.onCreateOptionsMenu(menu));
}
```

Note that you will also need to define imports for `android.view.Menu` and `android.view.MenuInflater` for this to compile cleanly.

At this point, you can rebuild and reinstall the application. Click the [MENU] button, from either tab, to see the option menu with its icon:

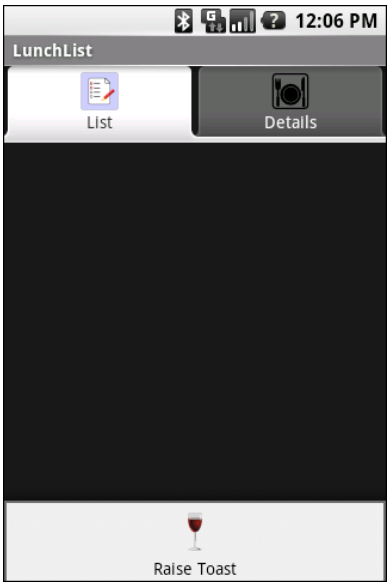


Figure 15. The LunchList option menu, displayed, with one menu choice

Step #4: Show the Notes as a Toast

Finally, we need to get control when the user selects the Raise Toast menu choice and display the notes in a `Toast`.

The problem is that, to do this, we need to know what `Restaurant` to show. So far, we have not been holding onto a specific `Restaurant` except when we needed it, such as when we populate the details form. Now, we need to know our current `Restaurant`, defined as the one visible in the detail form...which could be none, if we have not yet saved anything in the form.

To make all of this work, do the following:

1. Add another data member, `Restaurant current`, to hold the current `Restaurant`. Be sure to initialize it to null.
2. In `onSave` and `onListClick`, rather than declaring local `Restaurant` variables, use current to hold the `Restaurant` we are saving (in `onSave`) or have clicked on (in `onListClick`). You will need to change all references to the old `r` variable to be `current` in these two objects.

3. Add an import for `android.view.MenuItem`.
4. Add the following implementation of `onOptionsItemSelected()` to your `LunchList` class:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.toast) {
    String message="No restaurant selected";

    if (current!=null) {
      message=current.getNotes();
    }

    Toast.makeText(this, message, Toast.LENGTH_LONG).show();

    return(true);
  }

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

Note how we will either display "No restaurant selected" (if `current` is `null`) or the restaurant's notes, depending on our current state.

You can now rebuild and reinstall the application. Enter and save a `Restaurant`, with notes, then choose the Raise Toast option menu item, and you will briefly see your notes in a `Toast`:

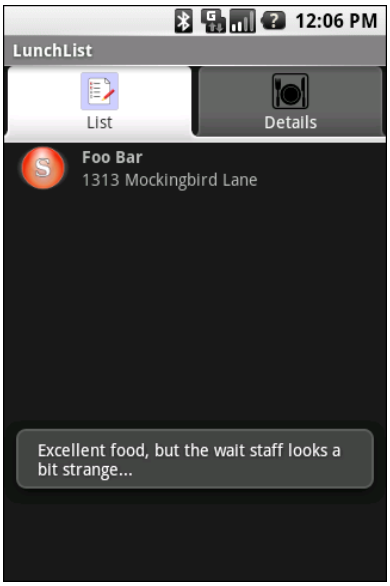


Figure 16. The Toast displayed, with some notes

The `LunchList` activity, as a whole, is shown below, incorporating all of the changes outlined in this tutorial:

```
package apt.tutorial;

import android.app.TabActivity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.view.ViewGroup;
import android.view.LayoutInflater;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ImageView;
import android.widget.ListView;
import android.widget.RadioGroup;
import android.widget.TabHost;
import android.widget.TextView;
import android.widget.Toast;
import java.util.ArrayList;
import java.util.List;

public class LunchList extends TabActivity {
  List<Restaurant> model=new ArrayList<Restaurant>();
```

```
  RestaurantAdapter adapter=null;
  EditText name=null;
  EditText address=null;
  EditText notes=null;
  RadioGroup types=null;
  Restaurant current=null;

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

    name=(EditText)findViewById(R.id.name);
    address=(EditText)findViewById(R.id.addr);
    notes=(EditText)findViewById(R.id.notes);
    types=(RadioGroup)findViewById(R.id.types);

    Button save=(Button)findViewById(R.id.save);

    save.setOnClickListener(onSave);

    ListView list=(ListView)findViewById(R.id.restaurants);

    adapter=new RestaurantAdapter();
    list.setAdapter(adapter);

    TabHost.TabSpec spec=getTabHost().newTabSpec("tag1");

    spec.setContent(R.id.restaurants);
    spec.setIndicator("List", getResources()
                              .getDrawable(R.drawable.list));
    getTabHost().addTab(spec);

    spec=getTabHost().newTabSpec("tag2");
    spec.setContent(R.id.details);
    spec.setIndicator("Details", getResources()
                                 .getDrawable(R.drawable.restaurant));
    getTabHost().addTab(spec);

    getTabHost().setCurrentTab(0);

    list.setOnItemClickListener(onListClick);
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    new MenuInflater(getApplication())
                                 .inflate(R.menu.option, menu);

    return(super.onCreateOptionsMenu(menu));
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
```

```
    if (item.getItemId()==R.id.toast) {
      String message="No restaurant selected";

      if (current!=null) {
        message=current.getNotes();
      }

      Toast.makeText(this, message, Toast.LENGTH_LONG).show();

      return(true);
    }

    return(super.onOptionsItemSelected(item));
  }

  private View.OnClickListener onSave=new View.OnClickListener() {
    public void onClick(View v) {
      current=new Restaurant();
      current.setName(name.getText().toString());
      current.setAddress(address.getText().toString());
      current.setNotes(notes.getText().toString());

      switch (types.getCheckedRadioButtonId()) {
        case R.id.sit_down:
          current.setType("sit_down");
          break;

        case R.id.take_out:
          current.setType("take_out");
          break;

        case R.id.delivery:
          current.setType("delivery");
          break;
      }

      adapter.add(current);
    }
  };

  private AdapterView.OnItemClickListener onListClick=new
AdapterView.OnItemClickListener() {
    public void onItemClick(AdapterView<?> parent,
                            View view, int position,
                            long id) {
      current=model.get(position);

      name.setText(current.getName());
      address.setText(current.getAddress());
      notes.setText(current.getNotes());

      if (current.getType().equals("sit_down")) {
        types.check(R.id.sit_down);
      }
```

```
    else if (current.getType().equals("take_out")) {
      types.check(R.id.take_out);
    }
    else {
      types.check(R.id.delivery);
    }

    getTabHost().setCurrentTab(1);
  }
};

class RestaurantAdapter extends ArrayAdapter<Restaurant> {
  RestaurantAdapter() {
    super(LunchList.this,
          android.R.layout.simple_list_item_1,
          model);
  }

  public View getView(int position, View convertView,
                      ViewGroup parent) {
    View row=convertView;
    RestaurantWrapper wrapper=null;

    if (row==null) {
      LayoutInflater inflater=getLayoutInflater();

      row=inflater.inflate(R.layout.row, null);
      wrapper=new RestaurantWrapper(row);
      row.setTag(wrapper);
    }
    else {
      wrapper=(RestaurantWrapper)row.getTag();
    }

    wrapper.populateFrom(model.get(position));

    return(row);
  }
}

class RestaurantWrapper {
  private TextView name=null;
  private TextView address=null;
  private ImageView icon=null;
  private View row=null;

  RestaurantWrapper(View row) {
    this.row=row;
  }

  void populateFrom(Restaurant r) {
    getName().setText(r.getName());
    getAddress().setText(r.getAddress());
```

```
      if (r.getType().equals("sit_down")) {
        getIcon().setImageResource(R.drawable.ball_red);
      }
      else if (r.getType().equals("take_out")) {
        getIcon().setImageResource(R.drawable.ball_yellow);
      }
      else {
        getIcon().setImageResource(R.drawable.ball_green);
      }
    }

    TextView getName() {
      if (name==null) {
        name=(TextView)row.findViewById(R.id.name);
      }

      return(name);
    }

    TextView getAddress() {
      if (address==null) {
        address=(TextView)row.findViewById(R.id.address);
      }

      return(address);
    }

    ImageView getIcon() {
      if (icon==null) {
        icon=(ImageView)row.findViewById(R.id.icon);
      }

      return(icon);
    }
  }
}
```

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Try using an `AlertDialog` instead of a `Toast` to display the message.
- Try adding a menu option to switch you between tabs. In particular, change the text and icon on the menu option to reflect the other tab (i.e., on the List tab, the menu should show "Details" and the details tab icon; on the Details tab, the menu should show "List" and the List tab icon).

- Try creating an `ErrorDialog` designed to display exceptions in a "pleasant" format to the end user. The `ErrorDialog` should also log the exceptions via `android.util.Log`. Use some sort of runtime exception (e.g., division by zero) for generating exceptions to pass to the dialog.

TUTORIAL 8

Sitting in the Background

In this tutorial, we will simulate having the `LunchList` do some background processing in a secondary thread, updating the user interface via a progress bar. While all of these tutorials are somewhat contrived, this one will be more contrived than most, as there is not much we are really able to do in a `LunchList` that would even require long processing in a background thread. So, please forgive us if this tutorial is a bit goofy.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `07-MenusMessages` edition of `LunchList` to use as a starting point.

Step #1: Create the Work Method

The theory of this demo is that we have something that takes a long time, and we want to have that work done in a background thread and update a `ProgressBar` along the way. So, the first step is to build something that will run a long time.

To do that, first, implement a `doSomeLongWork()` method on `LunchList` as follows:

```
private void doSomeLongWork(final int incr) {
  SystemClock.sleep(250);  // should be something more useful!
}
```

Here, we sleep for 250 milliseconds, simulating doing some meaningful work.

Then, create a private `Runnable` in `LunchList` that will fire off `doSomeLongWork()` a number of times, as follows:

```
private Runnable longTask=new Runnable() {
  public void run() {
    for (int i=0;i<20;i++) {
      doSomeLongWork(5);
    }
  }
};
```

Here, we just loop 20 times, so the overall background thread will run for 5 seconds.

Step #2: Fork the Thread from the Menu

Next, we need to arrange to do this (fake) long work at some point. The easiest way to do that is add another menu choice. Update the `LunchList/res/menu/option.xml` file to look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/toast"
    android:title="Raise Toast"
    android:icon="@drawable/toast"
  />
  <item android:id="@+id/run"
    android:title="Run Long Task"
    android:icon="@drawable/run"
  />
</menu>
```

This requires a graphic image in `LunchList/res/drawable/run.png` – find something that you can use that is around 32px high.

Since the menu item is in the menu XML, we do not need to do anything special to display the item – it will just be added to the menu automatically. We do, however, need to arrange to do something useful when the menu choice is chosen. So, update `onOptionsItemSelected()` in `LunchList` to look like the following:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.toast) {
    String message="No restaurant selected";

    if (current!=null) {
      message=current.getNotes();
    }

    Toast.makeText(this, message, Toast.LENGTH_LONG).show();

    return(true);
  }
  else if (item.getItemId()==R.id.run) {
    new Thread(longTask).start();
  }

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

You are welcome to recompile, reinstall, and run the application. However, since our background thread does not do anything visible at the moment, all you will see that is different is the new menu item:

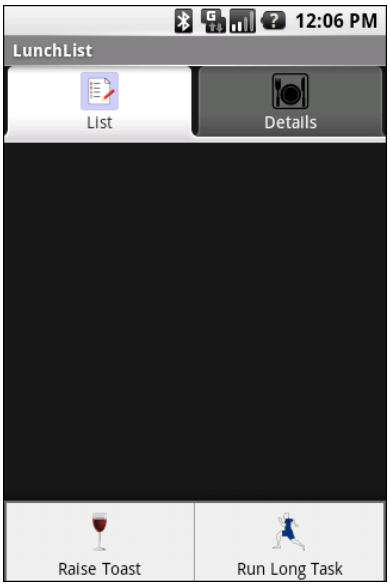


Figure 17. The Run Long Task menu item

Step #3: Add the Progress Bar

Now, we want to add a `ProgressBar` to our UI. We will tuck the bar inside the top of the list of `Restaurant` objects...but set it to be initially "gone" (`android:visibility = "gone"`). This means it will take up no space in our layout, until such time as we make it visible.

So, modify the `LunchList/res/layout/main.xml` to look like:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <TabHost android:id="@android:id/tabhost"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <TabWidget android:id="@android:id/tabs"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
    />
    <FrameLayout android:id="@android:id/tabcontent"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
```

```
    android:paddingTop="62px"
    >
    <LinearLayout
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      >
      <ProgressBar android:id="@+id/progress"
        style="?android:attr/progressBarStyleHorizontal"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:visibility="gone"
      />
      <ListView android:id="@+id/restaurants"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
      />
    </LinearLayout>
    <TableLayout android:id="@+id/details"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:stretchColumns="1"
      android:paddingTop="4px"
      >
      <TableRow>
        <TextView android:text="Name:" />
        <EditText android:id="@+id/name" />
      </TableRow>
      <TableRow>
        <TextView android:text="Address:" />
        <EditText android:id="@+id/addr" />
      </TableRow>
      <TableRow>
        <TextView android:text="Type:" />
        <RadioGroup android:id="@+id/types">
          <RadioButton android:id="@+id/take_out"
            android:text="Take-Out"
          />
          <RadioButton android:id="@+id/sit_down"
            android:text="Sit-Down"
          />
          <RadioButton android:id="@+id/delivery"
            android:text="Delivery"
          />
        </RadioGroup>
      </TableRow>
      <TableRow>
        <TextView android:text="Notes:" />
        <EditText android:id="@+id/notes"
          android:singleLine="false"
          android:gravity="top"
          android:lines="2"
          android:scrollHorizontally="false"
          android:maxLines="2"
          android:maxWidth="200sp"
```

```
          />
        </TableRow>
        <Button android:id="@+id/save"
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="Save"
        />
      </TableLayout>
    </FrameLayout>
  </TabHost>
</LinearLayout>
```

Step #4: Manage the Progress Bar

Finally, we need to actually make use of the `ProgressBar`. This involves making it visible when we start our long-running task, updating it as the task proceeds, and hiding it again when the task is complete.

First, create a `ProgressBar` data member named progress and find it in our layout, just like we find our `EditText` widgets and so on.

Next, make it visible by updating `onOptionsItemSelected()` to show it:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.toast) {
    String message="No restaurant selected";

    if (current!=null) {
      message=current.getNotes();
    }

    Toast.makeText(this, message, Toast.LENGTH_LONG).show();

    return(true);
  }
  else if (item.getItemId()==R.id.run) {
    progress.setVisibility(View.VISIBLE);
    new Thread(longTask).start();

    return(true);
  }
```

Notice the extra line that makes progress visible.

Then, we need to update the progress bar on each pass, so make this change to `doSomeLongWork()`:

```
private void doSomeLongWork(final int incr) {
  runOnUiThread(new Runnable() {
    public void run() {
      progress.incrementProgressBy(incr);
    }
  });

  SystemClock.sleep(250);  // should be something more useful!
}
```

Notice how we use `runOnUiThread()` to make sure our progress bar update occurs on the UI thread.

Finally, we need to hide the progress bar when we are done, so make this change to our `longTask Runnable`:

```
private Runnable longTask=new Runnable() {
  public void run() {
    for (int i=0;i<20;i++) {
      doSomeLongWork(5);
    }

    runOnUiThread(new Runnable() {
      public void run() {
        progress.setVisibility(View.GONE);
        progress.setProgress(0);
      }
    });
  }
};
```

Notice how we also reset the progress bar via `setProgress(0);` so that it is ready for reuse.

At this point, you can rebuild, reinstall, and run the application. When you choose the Run Long Task menu item, you will see the progress bar appear for five seconds, progressively updated as the "work" gets done:

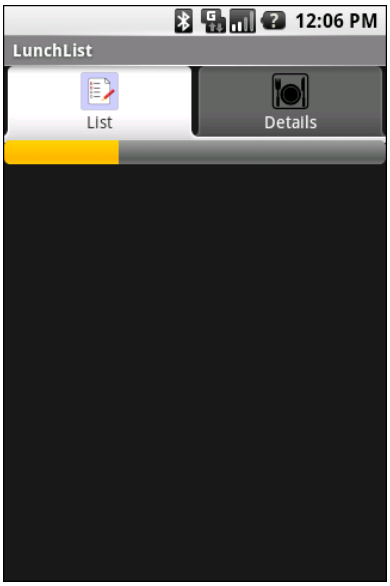


Figure 18. The progress bar in action

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Have the background thread also update some UI element when the work is completed, beyond dismissing the progress bar. Make sure you arrange to update the UI on the UI thread!
- Instead of using `Activity#runOnUiThread()`, try using a `Handler` for communication between the background thread and the UI thread.
- Instead of starting a `Thread` from the menu choice, have the `Thread` be created in `onCreate()` and have it monitor a `LinkedBlockingQueue` (from `java.util.concurrent`) as a source of work to be done. Create a `FakeJob` that does what our current long-running method does, and a `KillJob` that causes the `Thread` to fall out of its queue-monitoring loop.

Life and Times

In this tutorial, we will make our background task take a bit longer, then arrange to pause the background work when we start up another activity and restart the background work when our activity regains control. This pattern – stopping unnecessary background work when the activity is paused – is a good design pattern and is not merely something used for a tutorial.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `08-Threads` edition of `LunchList` to use as a starting point.

Step #1: Lengthen the Background Work

First, let us make the background work take a bit longer, so we have a bigger "window" in which to test whether our pause-and-resume logic works. It is also helpful, in our case, to synchronize our loop with our progress bar, so rather than counting `0` to `50` by `1`, we should count from `0` to `100` by `2`, so the loop counter and `ProgressBar` progress are the same.

In the `longTask Runnable`, change the loop to look like this:

```
for (int i=progress.getProgress();
    i<100;
    i+=2) {
  doSomeLongWork(2);
}
```

Step #2: Pause in onPause()

Now, we need to arrange to have our thread stop running when the activity is paused (e.g., some other activity has taken over the screen). Since threads are relatively cheap to create and destroy, we can simply have our current running thread stop and start a fresh one, if needed, in `onResume()`.

While there are some deprecated methods on `Thread` to try to forcibly terminate them, it is generally better to let the `Thread` stop itself by falling out of whatever processing loop it is in. So, what we want to do is let the background thread know the activity is not active.

To do this, first import `java.util.concurrent.atomic.AtomicBoolean` in `LunchList` and add an `AtomicBoolean` data member named `isActive`, initially set to `true` (`new AtomicBoolean(true);`).

Then, in the `longTask Runnable`, change the loop to also watch for the state of `isActive`, falling out of the loop if the activity is no longer active:

```
for (int i=progress.getProgress();
    i<100 && isActive.get();
    i+=2) {
  doSomeLongWork(2);
}
```

Finally, implement `onPause()` to update the state of `isActive`:

```
@Override
public void onPause() {
  super.onPause();

  isActive.set(false);
}
```

Note how we chain to the superclass in `onPause()` – if we fail to do this, we will get a runtime error.

With this implementation, our background thread will run to completion or until `isActive` is `false`, whichever comes first.

Step #3: Resume in onResume()

Now, we need to restart our thread if it is needed. It will be needed if the progress of the `ProgressBar` is greater than `0`, indicating we were in the middle of our background work when our activity was so rudely interrupted.

So, add the following implementation of `onResume()`:

```
@Override
public void onResume() {
  super.onResume();

  isActive.set(true);

  if (progress.getProgress()>0) {
    startWork();
  }
}
```

This assumes we have pulled out our thread-starting logic into a `startWork()` method, which you should implement as follows:

```
private void startWork() {
  progress.setVisibility(View.VISIBLE);
  new Thread(longTask).start();
}
```

And you can change our menu handler to also use `startWork()`:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.toast) {
    String message="No restaurant selected";

    if (current!=null) {
      message=current.getNotes();
    }
```

```
    Toast.makeText(this, message, Toast.LENGTH_LONG).show();

    return(true);
  }
  else if (item.getItemId()==R.id.run) {
    startWork();

    return(true);
  }
```

Finally, we need to not reset and hide the `ProgressBar` when our background thread ends if it ends because our activity is not active. Otherwise, we will never restart it, since the bar's progress will be reset to `0` every time. So, change `longTask` one more time, to look like this:

```
private Runnable longTask=new Runnable() {
  public void run() {
    for (int i=progress.getProgress();
        i<100 && isActive.get();
        i+=2) {
      doSomeLongWork(2);
    }

    if (isActive.get()) {
      runOnUiThread(new Runnable() {
        public void run() {
          progress.setVisibility(View.GONE);
          progress.setProgress(0);
        }
      });
    }
  }
};
```

What this does is reset the progress bar only if we are active when the work is complete, so we are ready for the next round of work. If we are inactive, and fell out of our loop for that reason, we leave the progress bar as-is.

At this point, recompile and reinstall the application. To test this feature:

1. Use the [MENU] button to run the long task.
2. While it is running, click the green phone button on the emulator (lower-left corner of the "phone"). This will bring up the call log activity and, as a result, pause our `LunchList` activity.

3. After a while, click the back button – you should see the `LunchList` resuming the background work from the point where it left off.

Here is the full `LunchList` implementation, including the changes shown above:

```
package apt.tutorial;

import android.app.TabActivity;
import android.os.Bundle;
import android.os.SystemClock;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.view.ViewGroup;
import android.view.LayoutInflater;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ImageView;
import android.widget.ListView;
import android.widget.ProgressBar;
import android.widget.RadioGroup;
import android.widget.TabHost;
import android.widget.TextView;
import android.widget.Toast;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.AtomicBoolean;

public class LunchList extends TabActivity {
  List<Restaurant> model=new ArrayList<Restaurant>();
  RestaurantAdapter adapter=null;
  EditText name=null;
  EditText address=null;
  EditText notes=null;
  RadioGroup types=null;
  Restaurant current=null;
  ProgressBar progress=null;
  AtomicBoolean isActive=new AtomicBoolean(true);

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

    name=(EditText)findViewById(R.id.name);
    address=(EditText)findViewById(R.id.addr);
    notes=(EditText)findViewById(R.id.notes);
    types=(RadioGroup)findViewById(R.id.types);
```

```
    progress=(ProgressBar)findViewById(R.id.progress);

    Button save=(Button)findViewById(R.id.save);

    save.setOnClickListener(onSave);

    ListView list=(ListView)findViewById(R.id.restaurants);

    adapter=new RestaurantAdapter();
    list.setAdapter(adapter);

    TabHost.TabSpec spec=getTabHost().newTabSpec("tag1");

    spec.setContent(R.id.restaurants);
    spec.setIndicator("List", getResources()
                              .getDrawable(R.drawable.list));
    getTabHost().addTab(spec);

    spec=getTabHost().newTabSpec("tag2");
    spec.setContent(R.id.details);
    spec.setIndicator("Details", getResources()
                                 .getDrawable(R.drawable.restaurant));
    getTabHost().addTab(spec);

    getTabHost().setCurrentTab(0);

    list.setOnItemClickListener(onListClick);
  }

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

    isActive.set(false);
  }

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

    isActive.set(true);

    if (progress.getProgress()>0) {
      startWork();
    }
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    new MenuInflater(getApplication())
                                 .inflate(R.menu.option, menu);

    return(super.onCreateOptionsMenu(menu));
  }
```

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.toast) {
    String message="No restaurant selected";

    if (current!=null) {
      message=current.getNotes();
    }

    Toast.makeText(this, message, Toast.LENGTH_LONG).show();

    return(true);
  }
  else if (item.getItemId()==R.id.run) {
    startWork();

    return(true);
  }

  return(super.onOptionsItemSelected(item));
}

private void startWork() {
  progress.setVisibility(View.VISIBLE);
  new Thread(longTask).start();
}

private void doSomeLongWork(final int incr) {
  runOnUiThread(new Runnable() {
    public void run() {
      progress.incrementProgressBy(incr);
    }
  });

  SystemClock.sleep(250);  // should be something more useful!
}

private View.OnClickListener onSave=new View.OnClickListener() {
  public void onClick(View v) {
    current=new Restaurant();
    current.setName(name.getText().toString());
    current.setAddress(address.getText().toString());
    current.setNotes(notes.getText().toString());

    switch (types.getCheckedRadioButtonId()) {
      case R.id.sit_down:
        current.setType("sit_down");
        break;

      case R.id.take_out:
        current.setType("take_out");
        break;
```

```
        case R.id.delivery:
          current.setType("delivery");
          break;
      }

      adapter.add(current);
    }
  };

  private AdapterView.OnItemClickListener onListClick=new
AdapterView.OnItemClickListener() {
    public void onItemClick(AdapterView<?> parent,
                            View view, int position,
                            long id) {
      current=model.get(position);

      name.setText(current.getName());
      address.setText(current.getAddress());
      notes.setText(current.getNotes());

      if (current.getType().equals("sit_down")) {
        types.check(R.id.sit_down);
      }
      else if (current.getType().equals("take_out")) {
        types.check(R.id.take_out);
      }
      else {
        types.check(R.id.delivery);
      }

      getTabHost().setCurrentTab(1);
    }
  };

  private Runnable longTask=new Runnable() {
    public void run() {
      for (int i=progress.getProgress();
          i<100 && isActive.get();
          i+=2) {
        doSomeLongWork(2);
      }

      if (isActive.get()) {
        runOnUiThread(new Runnable() {
          public void run() {
            progress.setVisibility(View.GONE);
            progress.setProgress(0);
          }
        });
      }
    }
  };

  class RestaurantAdapter extends ArrayAdapter<Restaurant> {
```

```
  RestaurantAdapter() {
    super(LunchList.this,
          android.R.layout.simple_list_item_1,
          model);
  }

  public View getView(int position, View convertView,
                      ViewGroup parent) {
    View row=convertView;
    RestaurantWrapper wrapper=null;

    if (row==null) {
      LayoutInflater inflater=getLayoutInflater();

      row=inflater.inflate(R.layout.row, null);
      wrapper=new RestaurantWrapper(row);
      row.setTag(wrapper);
    }
    else {
      wrapper=(RestaurantWrapper)row.getTag();
    }

    wrapper.populateFrom(model.get(position));

    return(row);
  }
}

class RestaurantWrapper {
  private TextView name=null;
  private TextView address=null;
  private ImageView icon=null;
  private View row=null;

  RestaurantWrapper(View row) {
    this.row=row;
  }

  void populateFrom(Restaurant r) {
    getName().setText(r.getName());
    getAddress().setText(r.getAddress());

    if (r.getType().equals("sit_down")) {
      getIcon().setImageResource(R.drawable.ball_red);
    }
    else if (r.getType().equals("take_out")) {
      getIcon().setImageResource(R.drawable.ball_yellow);
    }
    else {
      getIcon().setImageResource(R.drawable.ball_green);
    }
  }

  TextView getName() {
```

```
      if (name==null) {
        name=(TextView)row.findViewById(R.id.name);
      }

      return(name);
    }

    TextView getAddress() {
      if (address==null) {
        address=(TextView)row.findViewById(R.id.address);
      }

      return(address);
    }

    ImageView getIcon() {
      if (icon==null) {
        icon=(ImageView)row.findViewById(R.id.icon);
      }

      return(icon);
    }
  }
}
```

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Have the progress bar position be persisted via `onSaveInstanceState()`. When the activity is started in `onCreate()`, see if the background work was in progress when the activity was shut down (i.e., progress bar further than `0`), and restart the background thread immediately if it was. To test this, you can press `<Ctrl>-<F12>` to simulate opening the keyboard and rotating the screen – by default, this causes your activity to be destroyed and recreated, with `onSaveInstanceState()` called along the way.
- Try moving the pause/resume logic to `onStop()` and `onStart()`.

A Few Good Resources

We have already used many types of resources in the preceding tutorials. After reviewing what we have used so far, we set up an alternate layout for our `LunchList` activity to be used when the activity is in landscape orientation instead of portrait.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `09-Lifecycle` edition of `LunchList` to use as a starting point.

Step #1: Review our Current Resources

Now that we have completed ten tutorials, this is a good time to recap what resources we have been using along the way. Right now, `LunchList` has:

- Seven icons in `LunchList/res/drawable/`, all PNGs
- Two XML files in `LunchList/res/layout/`, representing the main `LunchList` UI and the definition of each row
- One XML file in `LunchList/res/menu/`, containing our option menu definition

- The system-created `strings.xml` file in `LunchList/res/values/`, which presently just holds the name of our application:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">LunchList</string>
</resources>
```

Step #2: Create a Landscape Layout

In the emulator, with `LunchList` running and showing the details form, press `<Ctrl>-<F12>`. This simulates opening and closing the keyboard, causing the screen to rotate to landscape and portrait, respectively. Our current layout is not very good in landscape orientation:

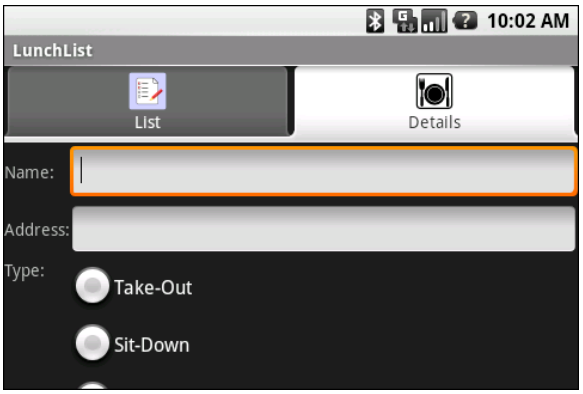


Figure 19. The LunchList in landscape orientation

So, let us come up with an alternative layout that will work better.

First, create a `LunchList/res/layout-land/` directory in your project. This will hold layout files that we wish to use when the device (or emulator) is in the landscape orientation.

Then, copy `LunchList/res/layout/main.xml` into `LunchList/res/layout-land/`, so we can start with the same layout we were using for portrait mode.

Then, change the layout to look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <TabHost android:id="@android:id/tabhost"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <TabWidget android:id="@android:id/tabs"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
    />
    <FrameLayout android:id="@android:id/tabcontent"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:paddingTop="62px"
      >
      <LinearLayout
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        >
        <ProgressBar android:id="@+id/progress"
          style="?android:attr/progressBarStyleHorizontal"
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:visibility="gone"
        />
        <ListView android:id="@+id/restaurants"
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
        />
      </LinearLayout>
      <TableLayout android:id="@+id/details"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:stretchColumns="1,3"
        android:paddingTop="4px"
        >
        <TableRow>
          <TextView
            android:text="Name:"
            android:paddingRight="2px"
          />
          <EditText
            android:id="@+id/name"
            android:maxWidth="140sp"
          />
          <TextView
            android:text="Address:"
            android:paddingLeft="2px"
            android:paddingRight="2px"
          />
          <EditText
            android:id="@+id/addr"
            android:maxWidth="140sp"
```

```
          />
        </TableRow>
        <TableRow>
          <TextView android:text="Type:" />
          <RadioGroup android:id="@+id/types">
            <RadioButton android:id="@+id/take_out"
              android:text="Take-Out"
            />
            <RadioButton android:id="@+id/sit_down"
              android:text="Sit-Down"
            />
            <RadioButton android:id="@+id/delivery"
              android:text="Delivery"
            />
          </RadioGroup>
          <TextView
            android:text="Notes:"
            android:paddingLeft="2px"
          />
          <LinearLayout
            android:layout_width="fill_parent"
            android:layout_height="fill_parent"
            android:orientation="vertical"
            >
            <EditText android:id="@+id/notes"
              android:singleLine="false"
              android:gravity="top"
              android:lines="3"
              android:scrollHorizontally="false"
              android:maxLines="3"
              android:maxWidth="140sp"
              android:layout_width="fill_parent"
              android:layout_height="wrap_content"
            />
            <Button android:id="@+id/save"
              android:layout_width="fill_parent"
              android:layout_height="wrap_content"
              android:text="Save"
            />
          </LinearLayout>
        </TableRow>
      </TableLayout>
    </FrameLayout>
  </TabHost>
</LinearLayout>
```

In this revised layout, we:

- Switched to four columns in our table, with columns #1 and #3 as stretchable
- Put the name and address labels and fields on the same row

- Put the type, notes, and Save button on the same row, with the notes and Save button stacked via a `LinearLayout`
- Made the notes three lines instead of two, since we have the room
- Fixed the maximum width of the `EditText` widgets to 140 scaled pixels (`sp`), so they do not automatically grow outlandishly large if we type a lot
- Added a bit of padding in places to make the placement of the labels and fields look a bit better

If you rebuild and reinstall the application, then run it in landscape mode, you will see a form that looks like this:

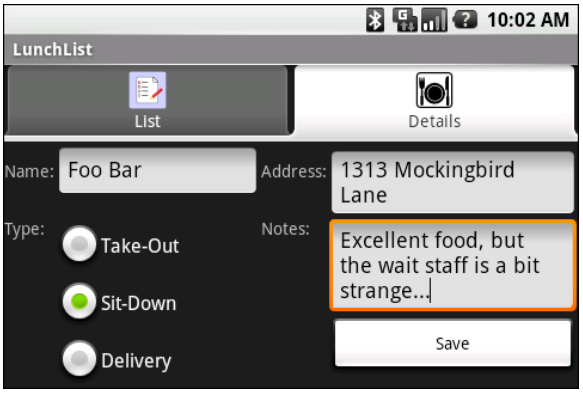


Figure 20. The LunchList in landscape orientation, revised

Note that we did not create a `LunchList/res/layout-land/` edition of our row layout (`row.xml`). Android, upon not finding one in `LunchList/res/layout-land/`, will fall back to the one in `LunchList/res/layout-land/`. Since we do not really need our row to change, we can leave it as-is.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Find some other icons to use and create a `LunchList/res/drawable-land` directory with the replacement icons, using the same names as

found in `LunchList/res/drawable`. See if exposing the keyboard swaps the icons as well as the layouts.

- Change the text of the labels in our main layout file to be string resources. You will need to add those values to `LunchList/res/values/strings.xml` and reference them in `LunchList/res/layout/main.xml`.
- Use `onSaveInstanceState()` to save the current contents of the details form, and restore those contents in `onCreate()` if an instance state is available (e.g., after the screen was rotated). Note how this does not cover the list – you will still lose all existing `Restaurant` objects on a rotation event. However, in a later tutorial, we will move that data to the database, which will solve that problem.

The Restaurant Store

In this tutorial, we will create a database and table for holding our `Restaurant` data and switch from our `ArrayAdapter` to a `CursorAdapter`, to make use of that database. This will allow our `Restaurant` objects to persist from run to run of `LunchList`.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `10-Resources` edition of `LunchList` to use as a starting point.

Step #1: Create a Stub SQLiteOpenHelper

First, we need to be able to define what our database name is, what the schema is for the table for our Restaurant objects, etc. That is best wrapped up in a `SQLiteOpenHelper` implementation.

So, create `LunchList/src/apt/tutorial/RestaurantSQLiteHelper.java`, and enter in the following code:

```
package apt.tutorial;

import android.content.Context;
```

```
import android.database.SQLException;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteQueryBuilder;

class RestaurantSQLiteHelper extends SQLiteOpenHelper {
  private static final String DATABASE_NAME="lunchlist.db";
  private static final int SCHEMA_VERSION=1;

  public RestaurantSQLiteHelper(Context context) {
    super(context, DATABASE_NAME, null, SCHEMA_VERSION);
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
  }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
  }
}
```

This says that our database name is `lunchlist.db`, we are using the first version of the schema...and not much else. However, the project should still compile cleanly after adding this class.

Step #2: Manage our Schema

Next, we need to flesh out the `onCreate()` and `onUpgrade()` methods in `RestaurantSQLiteHelper`, to actually create the schema we want.

To do this, add an import for `android.database.Cursor` and use the following implementation of `onCreate()`:

```
@Override
public void onCreate(SQLiteDatabase db) {
  Cursor c=db.rawQuery("SELECT name FROM sqlite_master WHERE type='table' AND
name='restaurants'", null);

  try {
    if (c.getCount()==0) {
      db.execSQL("CREATE TABLE restaurants (_id INTEGER PRIMARY KEY
AUTOINCREMENT, name TEXT, address TEXT, type TEXT, notes TEXT);");
    }
  }
  finally {
    c.close();
```

```
  }
}
```

We are seeing if there already is a `restaurant` table, and if not, executing a SQL statement to create it.

For `onUpgrade()`, we can "cheat" and simply decide to throw out our existing `restaurant` table and recreate it. So, enter the following code as the implemenation of `onUpgrade()`:

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
  android.util.Log.w("LunchList", "Upgrading database, which will destroy all
old data");
  db.execSQL("DROP TABLE IF EXISTS restaurants");
  onCreate(db);
}
```

In a production system, of course, we would want to make a temporary table, copy our current data to it, fix up the real table's schema, then migrate the data back.

Step #3: Open and Close the Database

In order to really use the database, though, we need to open and close access to it from `LunchList`.

First, add an import to `android.database.sqlite.SQLiteDatabase` to `LunchList` and a `SQLiteDatabase` data member named `db`, initially set to `null`.

Then, in `onCreate()` in `LunchList`, set up db to be a writeable `SQLiteDatabase` instance by adding the following code, somewhere near the top of the method:

```
db=(new RestaurantSQLiteHelper(this))
                                .getWritableDatabase();
```

Finally, implement `onDestroy()` on `LunchList` as follows:

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

  db.close();
}
```

All we do in `onDestroy()`, besides chain to the superclass, is close the database we opened in `onCreate()`.

At this point, we have an open handle to the database which we can use during the lifetime of our `LunchList` activity.

Step #4: Save a Restaurant to the Database

We need to arrange to insert a `Restaurant` object into the database when it is saved, so our `Restaurant` objects persist from run to run of `LunchList`. Let us add this as a method on `Restaurant`, so the model class knows how to persist itself to the database.

First, add `import` statements for `android.content.ContentValues` and `android.database.sqlite.SQLiteDatabase` to `Restaurant`.

Then, implement `save()` on `Restaurant` as follows:

```
void save(SQLiteDatabase db) {
  ContentValues cv=new ContentValues();

  cv.put("name", name);
  cv.put("address", address);
  cv.put("type", type);
  cv.put("notes", notes);
  db.insert("restaurants", "name", cv);
}
```

With this code, we convert our four data members into entries in a `ContentValues` and tell the `SQLiteDatabase` to insert it into the database.

Finally, we need to actually call `save()` on the `Restaurant` at the appropriate time. Right now, our Save button adds the `Restaurant` to our

RestaurantAdapter – now, we need it to tell the Restaurant to persist itself to the database. So, modify the onSave object in LunchList to look like this:

```
private View.OnClickListener onSave=new View.OnClickListener() {
  public void onClick(View v) {
    current=new Restaurant();
    current.setName(name.getText().toString());
    current.setAddress(address.getText().toString());
    current.setNotes(notes.getText().toString());

    switch (types.getCheckedRadioButtonId()) {
      case R.id.sit_down:
        current.setType("sit_down");
        break;

      case R.id.take_out:
        current.setType("take_out");
        break;

      case R.id.delivery:
        current.setType("delivery");
        break;
    }

    current.save(db);
  }
};
```

The only change is in the final statement, replacing what was there with the call to save().

Step #5: Get the List of Restaurants from the Database

This puts Restaurant objects into the database. Presumably, it would be useful to get them back out sometime. Hence, we need some logic that can query the database and return a Cursor with columnar data from our restaurant table.

To do this, add the following method to Restaurant:

```
static Cursor getAll(SQLiteDatabase db) {
  return(db.rawQuery("SELECT * FROM restaurants ORDER BY name",
                     null));
}
```

Step #6: Change our Adapter

Of course, our existing `RestaurantAdapter` extends `ArrayAdapter` and cannot use a `Cursor` very effectively. So, we need to change our `RestaurantAdapter` into something that can use a `Cursor`...such as a `CursorAdapter`.

A `CursorAdapter` does not use `getView()`, but rather `bindView()` and `newView()`. The `newView()` method handles the case where we need to inflate a new row; `bindView()` is when we are recycling an existing row. So, our current `getView()` logic needs to be split between `bindView()` and `newView()`.

Replace our existing `RestaurantAdapter` implementation in `LunchList` with the following:

```
class RestaurantAdapter extends CursorAdapter {
  RestaurantAdapter(Cursor c) {
    super(LunchList.this, c);
  }

  @Override
  public void bindView(View row, Context ctxt,
                       Cursor c) {
    RestaurantWrapper wrapper=(RestaurantWrapper)row.getTag();

    wrapper.populateFrom(c);
  }

  @Override
  public View newView(Context ctxt, Cursor c,
                      ViewGroup parent) {
    LayoutInflater inflater=getLayoutInflater();

    View row=inflater.inflate(R.layout.row, null);
    RestaurantWrapper wrapper=new RestaurantWrapper(row);
    row.setTag(wrapper);

    wrapper.populateFrom(c);

    return(row);
  }
}
```

Then, you need to make use of this refined adapter, by changing the `model` in `LunchList` from an `ArrayList` to a `Cursor`. After you have changed that data

member, replace the current smarts to populate our `RestaurantAdapter` with the following:

```
model=Restaurant.getAll(db);
startManagingCursor(model);

adapter=new RestaurantAdapter(model);
list.setAdapter(adapter);
```

After getting the `Cursor` from `getAll()`, we call `startManagingCursor()`, so Android will deal with refreshing its contents if the activity is paused and resumed. Then, we hand the `Cursor` off to the `RestaurantAdapter`.

Step #7: Clean Up Lingering ArrayList References

Since we changed our model in `LunchList` from an `ArrayList` to a `Cursor`, anything that still assumes an `ArrayList` will not work.

Notably, the `onListClick` listener object tries to obtain a `Restaurant` from the `ArrayList`. Now, we need to move the `Cursor` to the appropriate position and get a `Restaurant` from that.

First, add the following method to `Restaurant`:

```
Restaurant loadFrom(Cursor c) {
  name=c.getString(c.getColumnIndex("name"));
  address=c.getString(c.getColumnIndex("address"));
  type=c.getString(c.getColumnIndex("type"));
  notes=c.getString(c.getColumnIndex("notes"));

  return(this);
}
```

This is akin to our `save()` implementation in reverse, getting values out of the `Cursor` and loading them into the `Restaurant` object's data members.

Then, modify `onListClick` to take advantage of the new `loadFrom()` method:

```
private AdapterView.OnItemClickListener onListClick=new
AdapterView.OnItemClickListener() {
```

```
  public void onItemClick(AdapterView<?> parent,
                          View view, int position,
                          long id) {
    model.moveToPosition(position);
    current=new Restaurant().loadFrom(model);

    name.setText(current.getName());
    address.setText(current.getAddress());
    notes.setText(current.getNotes());

    if (current.getType().equals("sit_down")) {
      types.check(R.id.sit_down);
    }
    else if (current.getType().equals("take_out")) {
      types.check(R.id.take_out);
    }
    else {
      types.check(R.id.delivery);
    }

    getTabHost().setCurrentTab(1);
  }
};
```

At this point, you can recompile and reinstall your application. If you try using it, it will launch and you can save `Restaurant` objects to the database. However, you will find that the list of `Restaurant` objects will not update unless you exit and restart the `LunchList` activity.

Step #8: Refresh Our List

The reason the list does not update is because neither the `Cursor` nor the `CursorAdapter` realize that the database contents have changed when we save our `Restaurant`. To resolve this, add `model.requery();` immediately after the call to `save()` in the `onSave` object in `LunchList`. This causes the `Cursor` to reload its contents from the database, which in turn will cause the `CursorAdapter` to redisplay the list.

Rebuild and reinstall the application and try it out. You should have all the functionality you had before, with the added benefit of `Restaurant` objects living from run to run of `LunchList`.

Here is an implementation of `LunchList` that incorporates all of the changes shown in this tutorial:

```
package apt.tutorial;

import android.app.TabActivity;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.os.Bundle;
import android.os.SystemClock;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.view.ViewGroup;
import android.view.LayoutInflater;
import android.widget.AdapterView;
import android.widget.Button;
import android.widget.CursorAdapter;
import android.widget.EditText;
import android.widget.ImageView;
import android.widget.ListView;
import android.widget.ProgressBar;
import android.widget.RadioGroup;
import android.widget.TabHost;
import android.widget.TextView;
import android.widget.Toast;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.AtomicBoolean;

public class LunchList extends TabActivity {
  RestaurantAdapter adapter=null;
  EditText name=null;
  EditText address=null;
  EditText notes=null;
  RadioGroup types=null;
  Restaurant current=null;
  ProgressBar progress=null;
  AtomicBoolean isActive=new AtomicBoolean(true);
  SQLiteDatabase db=null;
  Cursor model=null;

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

    name=(EditText)findViewById(R.id.name);
    address=(EditText)findViewById(R.id.addr);
    notes=(EditText)findViewById(R.id.notes);
    types=(RadioGroup)findViewById(R.id.types);
```

```
    progress=(ProgressBar)findViewById(R.id.progress);

    db=(new RestaurantSQLiteHelper(this))
                                    .getWritableDatabase();

    Button save=(Button)findViewById(R.id.save);

    save.setOnClickListener(onSave);

    ListView list=(ListView)findViewById(R.id.restaurants);

    model=Restaurant.getAll(db);
    startManagingCursor(model);

    adapter=new RestaurantAdapter(model);
    list.setAdapter(adapter);

    TabHost.TabSpec spec=getTabHost().newTabSpec("tag1");

    spec.setContent(R.id.restaurants);
    spec.setIndicator("List", getResources()
                            .getDrawable(R.drawable.list));
    getTabHost().addTab(spec);

    spec=getTabHost().newTabSpec("tag2");
    spec.setContent(R.id.details);
    spec.setIndicator("Details", getResources()
                              .getDrawable(R.drawable.restaurant));
    getTabHost().addTab(spec);

    getTabHost().setCurrentTab(0);

    list.setOnItemClickListener(onListClick);
  }

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

    isActive.set(false);
  }

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

    isActive.set(true);

    if (progress.getProgress()>0) {
      startWork();
    }
  }

  @Override
```

```
  public void onDestroy() {
    super.onDestroy();

    model.close();
    db.close();
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    new MenuInflater(getApplication())
                                .inflate(R.menu.option, menu);

    return(super.onCreateOptionsMenu(menu));
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.toast) {
      String message="No restaurant selected";

      if (current!=null) {
        message=current.getNotes();
      }

      Toast.makeText(this, message, Toast.LENGTH_LONG).show();

      return(true);
    }
    else if (item.getItemId()==R.id.run) {
      startWork();

      return(true);
    }

    return(super.onOptionsItemSelected(item));
  }

  private void startWork() {
    progress.setVisibility(View.VISIBLE);
    new Thread(longTask).start();
  }

  private void doSomeLongWork(final int incr) {
    runOnUiThread(new Runnable() {
      public void run() {
        progress.incrementProgressBy(incr);
      }
    });

    SystemClock.sleep(250);  // should be something more useful!
  }

  private View.OnClickListener onSave=new View.OnClickListener() {
    public void onClick(View v) {
```

```
      current=new Restaurant();
      current.setName(name.getText().toString());
      current.setAddress(address.getText().toString());
      current.setNotes(notes.getText().toString());

      switch (types.getCheckedRadioButtonId()) {
        case R.id.sit_down:
          current.setType("sit_down");
          break;

        case R.id.take_out:
          current.setType("take_out");
          break;

        case R.id.delivery:
          current.setType("delivery");
          break;
      }

      current.save(db);
      model.requery();
    }
  };

  private AdapterView.OnItemClickListener onListClick=new
AdapterView.OnItemClickListener() {
    public void onItemClick(AdapterView<?> parent,
                            View view, int position,
                            long id) {
      model.moveToPosition(position);
      current=new Restaurant().loadFrom(model);

      name.setText(current.getName());
      address.setText(current.getAddress());
      notes.setText(current.getNotes());

      if (current.getType().equals("sit_down")) {
        types.check(R.id.sit_down);
      }
      else if (current.getType().equals("take_out")) {
        types.check(R.id.take_out);
      }
      else {
        types.check(R.id.delivery);
      }

      getTabHost().setCurrentTab(1);
    }
  };

  private Runnable longTask=new Runnable() {
    public void run() {
      for (int i=progress.getProgress();
          i<100 && isActive.get();
```

```
        i+=2) {
      doSomeLongWork(2);
    }

    if (isActive.get()) {
      runOnUiThread(new Runnable() {
        public void run() {
          progress.setVisibility(View.GONE);
          progress.setProgress(0);
        }
      });
    }
  }
};

class RestaurantAdapter extends CursorAdapter {
  RestaurantAdapter(Cursor c) {
    super(LunchList.this, c);
  }

  @Override
  public void bindView(View row, Context ctxt,
                       Cursor c) {
    RestaurantWrapper wrapper=(RestaurantWrapper)row.getTag();

    wrapper.populateFrom(c);
  }

  @Override
  public View newView(Context ctxt, Cursor c,
                      ViewGroup parent) {
    LayoutInflater inflater=getLayoutInflater();

    View row=inflater.inflate(R.layout.row, null);
    RestaurantWrapper wrapper=new RestaurantWrapper(row);
    row.setTag(wrapper);

    wrapper.populateFrom(c);

    return(row);
  }
}

class RestaurantWrapper {
  private TextView name=null;
  private TextView address=null;
  private ImageView icon=null;
  private View row=null;

  RestaurantWrapper(View row) {
    this.row=row;
  }

  void populateFrom(Cursor c) {
```

```
      getName().setText(c.getString(c.getColumnIndex("name")));
      getAddress().setText(c.getString(c.getColumnIndex("address")));

      String type=c.getString(c.getColumnIndex("type"));

      if (type.equals("sit_down")) {
        getIcon().setImageResource(R.drawable.ball_red);
      }
      else if (type.equals("take_out")) {
        getIcon().setImageResource(R.drawable.ball_yellow);
      }
      else {
        getIcon().setImageResource(R.drawable.ball_green);
      }
    }

    TextView getName() {
      if (name==null) {
        name=(TextView)row.findViewById(R.id.name);
      }

      return(name);
    }

    TextView getAddress() {
      if (address==null) {
        address=(TextView)row.findViewById(R.id.address);
      }

      return(address);
    }

    ImageView getIcon() {
      if (icon==null) {
        icon=(ImageView)row.findViewById(R.id.icon);
      }

      return(icon);
    }
  }
}
```

Similarly, here is an implementation of `Restaurant` that contains the modifications from this tutorial:

```
package apt.tutorial;

import android.content.ContentValues;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;

public class Restaurant {
```

```
  private String name="";
  private String address="";
  private String type="";
  private String notes="";

  static Cursor getAll(SQLiteDatabase db) {
    return(db.rawQuery("SELECT * FROM restaurants ORDER BY name",
                       null));
  }

  Restaurant loadFrom(Cursor c) {
    name=c.getString(c.getColumnIndex("name"));
    address=c.getString(c.getColumnIndex("address"));
    type=c.getString(c.getColumnIndex("type"));
    notes=c.getString(c.getColumnIndex("notes"));

    return(this);
  }

  public String getName() {
    return(name);
  }

  public void setName(String name) {
    this.name=name;
  }

  public String getAddress() {
    return(address);
  }

  public void setAddress(String address) {
    this.address=address;
  }

  public String getType() {
    return(type);
  }

  public void setType(String type) {
    this.type=type;
  }

  public String getNotes() {
    return(notes);
  }

  public void setNotes(String notes) {
    this.notes=notes;
  }

  public String toString() {
    return(getName());
  }
```

```
  void save(SQLiteDatabase db) {
    ContentValues cv=new ContentValues();

    cv.put("name", name);
    cv.put("address", address);
    cv.put("type", type);
    cv.put("notes", notes);
    db.insert("restaurants", "name", cv);
  }
}
```

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Add a menu choice that will allow the user to change the sort order to be by address instead of by name. The sort order is determined by the SQL query used in `getAll()` in `Restaurant`.
- Download the database off the emulator (or device) and examine it using a SQLite client program. You can use `adb pull` to download `/data/data/apt.tutorial/databases/lunchlist.db`, or use Eclipse or DDMS to browse the emulator graphically.
- Use `adb shell` and the `sqlite3` program built into the emulator to examine the database in the emulator itself, without downloading it.

Getting More Active

In this tutorial, we will add support for both creating new `Restaurant` objects and editing ones that were previously entered. Along the way, we will get rid of our tabs, splitting the application into two activities: one for the list, and one for the detail form.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `11-Database` edition of `LunchList` to use as a starting point.

Also, for this specific tutorial, since there is a lot of cutting and pasting, you may wish to save off a copy of your current work before starting in on the modifications, so you can clip code from the original and paste it where it is needed.

Step #1: Create a Stub Activity

The first thing we need to do is create an activity to serve as our detail form. In a flash of inspiration, let's call it `DetailForm`. So, create a `LunchList/src/apt/tutorial/DetailForm.java` file with the following content:

```
package apt.tutorial;

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

public class DetailForm extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
//    setContentView(R.layout.main);
  }
}
```

This is just a stub activity, except it has the `setContentView()` line commented out. That is because we do not want to use `main.xml`, as that is the layout for `LunchList`. Since we do not have another layout ready yet, we can just comment out the line. As we will see, this is perfectly legal, but it means the activity will have no UI.

Step #2: Launch the Stub Activity on List Click

Now, we need to arrange to display this activity when the user clicks on a `LunchList` list item, instead of flipping to the original detail form tab in `LunchList`.

First, we need to add `DetailForm` to the `AndroidManifest.xml` file, so it is recognized by the system as being an available activity. Change the manifest to look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="apt.tutorial"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:label="@string/app_name">
        <activity android:name=".LunchList"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".DetailForm">
        </activity>
```

```
    </application>
</manifest>
```

Notice the second `<activity>` element, referencing the `DetailForm` class. Also note that it does not need an `<intent-filter>`, since we will be launching it ourselves rather than expecting the system to launch it for us.

Then, we need to start this activity when the list item is clicked. That is handled by our `onListClick` listener object. So, replace our current implementation with the following:

```
private AdapterView.OnItemClickListener onListClick=new
AdapterView.OnItemClickListener() {
  public void onItemClick(AdapterView<?> parent,
                          View view, int position,
                          long id) {
    Intent i=new Intent(LunchList.this, DetailForm.class);

    startActivity(i);
  }
};
```

Here we create an Intent that points to our `DetailForm` and call `startActivity()`on that `Intent`.

At this point, you should be able to recompile and reinstall the application. If you run it and click on an item in the list, it will open up the empty `DetailForm`. From there, you can click the back button to return to the main `LunchList` activity.

Step #3: Move the Detail Form UI

Now, the shredding begins – we need to start moving our detail form smarts out of `LunchList` and its layout to `DetailForm`.

First, create a `LunchList/res/layout/detail_form.xml`, using the detail form from `LunchList/res/layout/main.xml` as a basis:

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
```

```
  android:layout_height="wrap_content"
  android:stretchColumns="1"
  >
  <TableRow>
    <TextView android:text="Name:" />
    <EditText android:id="@+id/name" />
  </TableRow>
  <TableRow>
    <TextView android:text="Address:" />
    <EditText android:id="@+id/addr" />
  </TableRow>
  <TableRow>
    <TextView android:text="Type:" />
    <RadioGroup android:id="@+id/types">
      <RadioButton android:id="@+id/take_out"
        android:text="Take-Out"
      />
      <RadioButton android:id="@+id/sit_down"
        android:text="Sit-Down"
      />
      <RadioButton android:id="@+id/delivery"
        android:text="Delivery"
      />
    </RadioGroup>
  </TableRow>
  <TableRow>
    <TextView android:text="Notes:" />
    <EditText android:id="@+id/notes"
      android:singleLine="false"
      android:gravity="top"
      android:lines="2"
      android:scrollHorizontally="false"
      android:maxLines="2"
      android:maxWidth="200sp"
    />
  </TableRow>
  <Button android:id="@+id/save"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Save"
  />
</TableLayout>
```

This is just the detail form turned into its own standalone layout file.

Next, uncomment the `setContentView()` call in `onCreate()` in `DetailForm` and have it load this layout:

```
setContentView(R.layout.detail_form);
```

Then, we need to add all our logic for accessing the various form widgets, plus an `onSave` listener for our Save button, plus all necessary imports.

Set the import list for `DetailForm` to be:

```
import android.app.Activity;
import android.database.sqlite.SQLiteDatabase;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.RadioGroup;
import android.widget.TextView;
```

Then, add the following data members to the `DetailForm` class:

```
EditText name=null;
EditText address=null;
EditText notes=null;
RadioGroup types=null;
Restaurant current=null;
```

Then, copy the widget finders and stuff from `LunchList` into `DetailForm`:

```
name=(EditText)findViewById(R.id.name);
address=(EditText)findViewById(R.id.addr);
notes=(EditText)findViewById(R.id.notes);
types=(RadioGroup)findViewById(R.id.types);

Button save=(Button)findViewById(R.id.save);

save.setOnClickListener(onSave);
```

Finally, add the `onSave` listener object with a subset of the implementation from `LunchList`:

```
private View.OnClickListener onSave=new View.OnClickListener() {
  public void onClick(View v) {
    current.setName(name.getText().toString());
    current.setAddress(address.getText().toString());
    current.setNotes(notes.getText().toString());

    switch (types.getCheckedRadioButtonId()) {
      case R.id.sit_down:
        current.setType("sit_down");
        break;
```

```
      case R.id.take_out:
        current.setType("take_out");
        break;

      case R.id.delivery:
        current.setType("delivery");
        break;
    }
  }
};
```

This should compile cleanly, but since we are not setting our `current Restaurant` object anywhere, it would crash if we tried using it.

Step #4: Clean Up the Original UI

We need to clean up `LunchList` and its layout to reflect the fact that we moved much of the logic over to `DetailForm`.

First, get rid of the tabs and the detail form from `LunchList/res/layout/main.xml`, leaving us with:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
  <ProgressBar android:id="@+id/progress"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:visibility="gone"
  />
  <ListView android:id="@+id/restaurants"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
  />
</LinearLayout>
```

Next, delete `LunchList/res/layout_land/main.xml`, as we will revisit landscape layouts in a later tutorial.

At present, `LunchList` extends `TabActivity`, which is no longer what we need. Change it to extend `Activity` instead. You may need to add an import for `android.app.Activity` if you do not already have one.

Finally, get rid of the code from `onCreate()` that sets up the tabs, since they are no longer needed. The lines to delete are:

```
TabHost.TabSpec spec=getTabHost().newTabSpec("tag1");

spec.setContent(R.id.restaurants);
spec.setIndicator("List", getResources()
                        .getDrawable(R.drawable.list));
getTabHost().addTab(spec);

spec=getTabHost().newTabSpec("tag2");
spec.setContent(R.id.details);
spec.setIndicator("Details", getResources()
                          .getDrawable(R.drawable.restaurant));
getTabHost().addTab(spec);

getTabHost().setCurrentTab(0);
```

Step #5: Pass the Restaurant _ID

Now, let's step back a bit and think about what we are trying to achieve.

We want to be able to use `DetailForm` for both adding new `Restaurant` objects and editing an existing `Restaurant`. `DetailForm` needs to be able to tell those two scenarios apart. Also, `DetailForm` needs to know which item is to be edited.

To achieve this, we will pass an "extra" in our Intent that launches `DetailForm`, containing the ID (`_id` column) of the `Restaurant` to edit. We will use this if the `DetailForm` was launched by clicking on an existing `Restaurant`. If `DetailForm` receives an `Intent` lacking our "extra", it will know to add a new `Restaurant`.

First, we need to define a name for this "extra", so add the following data member to `LunchList`:

```
public final static String ID_EXTRA="apt.tutorial._ID";
```

We use the `apt.tutorial` namespace to ensure our "extra" name will not collide with any names perhaps used by the Android system.

Next, we need to modify `onListClick` again to add this "extra":

```
private AdapterView.OnItemClickListener onListClick=new
AdapterView.OnItemClickListener() {
  public void onItemClick(AdapterView<?> parent,
                          View view, int position,
                          long id) {
    Intent i=new Intent(LunchList.this, DetailForm.class);

    i.putExtra(ID_EXTRA, String.valueOf(id));
    startActivity(i);
  }
};
```

The `_id` of the `Restaurant` happens to be provided to us as the fourth parameter to `onItemClick()`. We turn it into a `String` because `DetailForm` will want it in `String` format, as we will see shortly.

Next, add the following data member to `DetailForm`:

```
String restaurantId=null;
```

This will be `null` if we are adding a new `Restaurant` or the string form of the ID if we are editing an existing `Restaurant`.

Finally, add the following line to the end of `onCreate()` in `DetailForm`:

```
restaurantId=getIntent().getStringExtra(LunchList.ID_EXTRA);
```

This will pull out our "extra", or leave `restaurantId` as `null` if there is no such "extra".

Step #6: Load the Restaurant Into the Form

In the case where we are editing an existing `Restaurant`, we need to load that `Restaurant` object from the database, then load the `Restaurant` contents into the `DetailForm`.

First, we need to have access to the database in `DetailForm`. So, add another data member to `DetailForm`:

```
SQLiteDatabase db=null;
```

Then, actually get access to the database by adding these lines in `onCreate()` of `DetailForm`:

```
db=(new RestaurantSQLiteHelper(this))
                                .getWritableDatabase();
```

Since we opened the database, we need to close it again, so add an `onDestroy()` implementation to `DetailForm` as follows:

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

  db.close();
}
```

Now that we have a handle to the database, we need to load a `Restaurant` given its ID. We can add this as another static method on the `Restaurant` class, so the model/database bridge code remains in one spot. So, add the following method to `Restaurant`:

```
static Restaurant getById(String id, SQLiteDatabase db) {
  String[] args={id};
  Cursor c=db.rawQuery("SELECT * FROM restaurants WHERE _id=?",
                       args);

  c.moveToFirst();

  Restaurant result=new Restaurant().loadFrom(c);

  c.close();
```

```
  return(result);
}
```

Then, add the following lines to the bottom of `onCreate()` in `DetailForm`:

```
if (restaurantId==null) {
  current=new Restaurant();
}
else {
  load();
}
```

The code snippet above references a `load()` method, which we need to add to `DetailForm`, based off of code originally in `LunchList`:

```
private void load() {
  current=Restaurant.getById(restaurantId, db);

  name.setText(current.getName());
  address.setText(current.getAddress());
  notes.setText(current.getNotes());

  if (current.getType().equals("sit_down")) {
    types.check(R.id.sit_down);
  }
  else if (current.getType().equals("take_out")) {
    types.check(R.id.take_out);
  }
  else {
    types.check(R.id.delivery);
  }
}
```

The net is that, at the end of `onCreate()`, we will have a `Restaurant` object – either a brand-new instance or one loaded from the database. Also, if we loaded the `Restaurant` from the database, its current contents will be loaded into the form widgets.

Step #7: Add an "Add" Menu Option

We have most of the logic in place to edit existing `Restaurant` objects. However, we still need to add a menu item for adding a new `Restaurant`.

To do this, change `LunchList/res/menu/option.xml` to remove the `toast` option and add in one for `add`:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/add"
    android:title="Add"
    android:icon="@android:drawable/ic_menu_add"
  />
  <item android:id="@+id/run"
    android:title="Run Long Task"
    android:icon="@drawable/run"
  />
</menu>
```

Note that the `add` menu item references a built-in icon supplied by Android. There are many such icons, not completely documented by the Android project.

Now that we have the menu option, we need to adjust our menu handling to match. Replace the `onOptionsItemSelected()` implementation in `LunchList` with the following:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.add) {
    startActivity(new Intent(this, DetailForm.class));

    return(true);
  }
  else if (item.getItemId()==R.id.run) {
    startWork();

    return(true);
  }

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

Here, we launch the `DetailForm` activity without our "extra", signalling to `DetailForm` that it is to add a new `Restaurant`.

Step #8: Detail Form Supports Add and Edit

Last, but certainly not least, we need to have `DetailForm` properly do useful work when the Save button is clicked. Specifically, we need to either insert or update the database. It would also be nice if we dismissed the `DetailForm` at that point and returned to the main `LunchList` activity.

To accomplish this, we first need to add an `update()` method to `Restaurant` that can perform a database update:

```
void update(String id, SQLiteDatabase db) {
  ContentValues cv=new ContentValues();
  String[] args={id};

  cv.put("name", name);
  cv.put("address", address);
  cv.put("type", type);
  cv.put("notes", notes);
  db.update("restaurants", cv, "_id=?", args);
}
```

Then, we need to adjust our `onSave` listener object in `DetailForm` to call the right method (`save()` or `update()`) and `finish()` our activity:

```
private View.OnClickListener onSave=new View.OnClickListener() {
  public void onClick(View v) {
    current.setName(name.getText().toString());
    current.setAddress(address.getText().toString());
    current.setNotes(notes.getText().toString());

    switch (types.getCheckedRadioButtonId()) {
      case R.id.sit_down:
        current.setType("sit_down");
        break;

      case R.id.take_out:
        current.setType("take_out");
        break;

      case R.id.delivery:
        current.setType("delivery");
        break;
    }

    if (restaurantId==null) {
      current.save(db);
    }
```

```
    else {
      current.update(restaurantId, db);
    }

    finish();
  }
};
```

At this point, you should be able to recompile and reinstall the application. When you first bring up the application, it will no longer show the tabs:

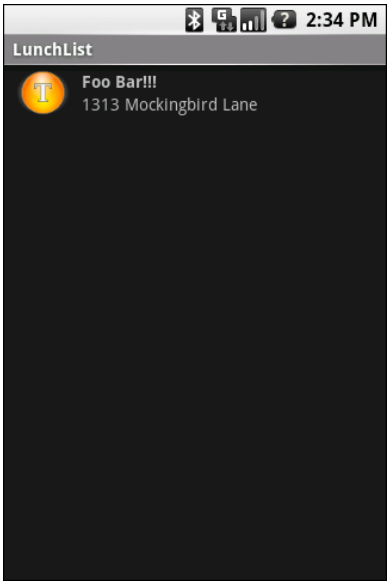


Figure 21. The new-and-improved LunchList

However, it will have an "add" menu option:

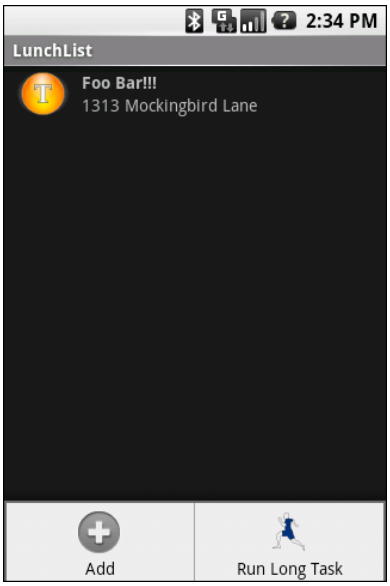


Figure 22. The LunchList option menu, with Add

If you choose the "add" menu option, it will bring up a blank `DetailForm`:

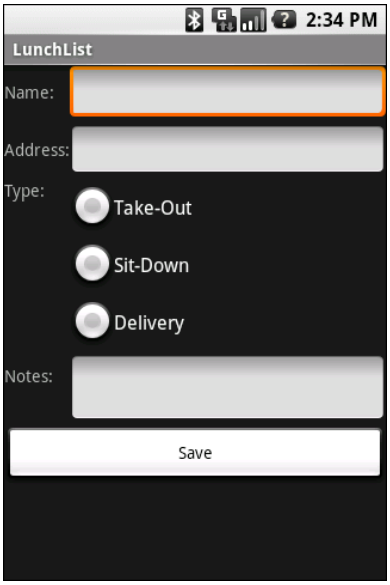


Figure 23. The DetailForm activity

If you fill out the form and click Save, it will return you to the `LunchList` and immediately shows the new `Restaurant` (courtesy of our using a managed `Cursor` in `LunchList`):

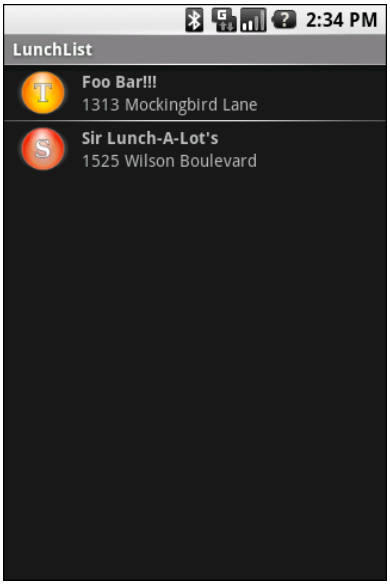


Figure 24. The LunchList with an added Restaurant

If you click an existing `Restaurant`, it will bring up the `DetailForm` for that object:

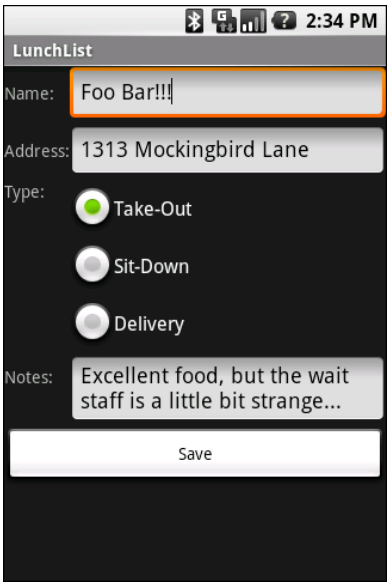


Figure 25. The DetailForm on an existing Restaurant

Making changes and clicking Save will update the database and list:

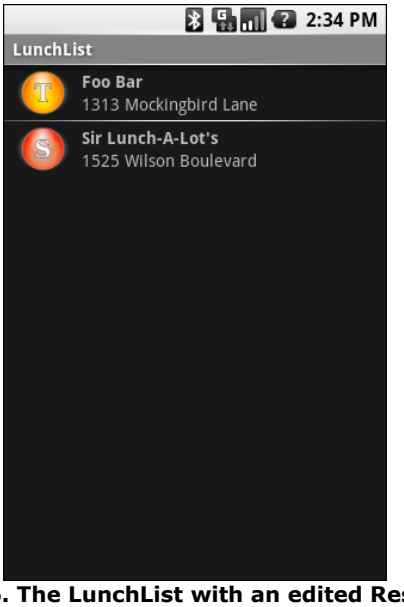


Figure 26. The LunchList with an edited Restaurant

Here is one implementation of `LunchList` that incorporates all of this tutorial's changes:

```
package apt.tutorial;

import android.app.Activity;
import android.content.Context;
import android.content.Intent;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.os.Bundle;
import android.os.SystemClock;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.view.ViewGroup;
import android.view.LayoutInflater;
import android.widget.AdapterView;
import android.widget.Button;
import android.widget.CursorAdapter;
import android.widget.EditText;
import android.widget.ImageView;
import android.widget.ListView;
import android.widget.ProgressBar;
import android.widget.RadioGroup;
import android.widget.TabHost;
import android.widget.TextView;
import android.widget.Toast;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.AtomicBoolean;

public class LunchList extends Activity {
  public final static String ID_EXTRA="apt.tutorial._ID";
  RestaurantAdapter adapter=null;
  ProgressBar progress=null;
  AtomicBoolean isActive=new AtomicBoolean(true);
  SQLiteDatabase db=null;
  Cursor model=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    progress=(ProgressBar)findViewById(R.id.progress);

    db=(new RestaurantSQLiteHelper(this))
                                  .getWritableDatabase();

    ListView list=(ListView)findViewById(R.id.restaurants);

    model=Restaurant.getAll(db);
```

```
    startManagingCursor(model);

    adapter=new RestaurantAdapter(model);
    list.setAdapter(adapter);
    list.setOnItemClickListener(onListClick);
  }

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

    isActive.set(false);
  }

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

    isActive.set(true);

    if (progress.getProgress()>0) {
      startWork();
    }
  }

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

    model.close();
    db.close();
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    new MenuInflater(getApplication())
                                .inflate(R.menu.option, menu);

    return(super.onCreateOptionsMenu(menu));
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.add) {
      startActivity(new Intent(this, DetailForm.class));

      return(true);
    }
    else if (item.getItemId()==R.id.run) {
      startWork();

      return(true);
    }
```

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

  private void startWork() {
    progress.setVisibility(View.VISIBLE);
    new Thread(longTask).start();
  }

  private void doSomeLongWork(final int incr) {
    runOnUiThread(new Runnable() {
      public void run() {
        progress.incrementProgressBy(incr);
      }
    });

    SystemClock.sleep(250);  // should be something more useful!
  }

  private AdapterView.OnItemClickListener onListClick=new
AdapterView.OnItemClickListener() {
    public void onItemClick(AdapterView<?> parent,
                            View view, int position,
                            long id) {
      Intent i=new Intent(LunchList.this, DetailForm.class);

      i.putExtra(ID_EXTRA, String.valueOf(id));
      startActivity(i);
    }
  };

  private Runnable longTask=new Runnable() {
    public void run() {
      for (int i=progress.getProgress();
          i<100 && isActive.get();
          i+=2) {
        doSomeLongWork(2);
      }

      if (isActive.get()) {
        runOnUiThread(new Runnable() {
          public void run() {
            progress.setVisibility(View.GONE);
            progress.setProgress(0);
          }
        });
      }
    }
  };

  class RestaurantAdapter extends CursorAdapter {
    RestaurantAdapter(Cursor c) {
      super(LunchList.this, c);
    }
```

```
  @Override
  public void bindView(View row, Context ctxt,
                       Cursor c) {
    RestaurantWrapper wrapper=(RestaurantWrapper)row.getTag();

    wrapper.populateFrom(c);
  }

  @Override
  public View newView(Context ctxt, Cursor c,
                      ViewGroup parent) {
    LayoutInflater inflater=getLayoutInflater();

    View row=inflater.inflate(R.layout.row, null);
    RestaurantWrapper wrapper=new RestaurantWrapper(row);
    row.setTag(wrapper);

    wrapper.populateFrom(c);

    return(row);
  }
}

class RestaurantWrapper {
  private TextView name=null;
  private TextView address=null;
  private ImageView icon=null;
  private View row=null;

  RestaurantWrapper(View row) {
    this.row=row;
  }

  void populateFrom(Cursor c) {
    getName().setText(c.getString(c.getColumnIndex("name")));
    getAddress().setText(c.getString(c.getColumnIndex("address")));

    String type=c.getString(c.getColumnIndex("type"));

    if (type.equals("sit_down")) {
      getIcon().setImageResource(R.drawable.ball_red);
    }
    else if (type.equals("take_out")) {
      getIcon().setImageResource(R.drawable.ball_yellow);
    }
    else {
      getIcon().setImageResource(R.drawable.ball_green);
    }
  }

  TextView getName() {
    if (name==null) {
      name=(TextView)row.findViewById(R.id.name);
    }
```

```
      return(name);
    }

    TextView getAddress() {
      if (address==null) {
        address=(TextView)row.findViewById(R.id.address);
      }

      return(address);
    }

    ImageView getIcon() {
      if (icon==null) {
        icon=(ImageView)row.findViewById(R.id.icon);
      }

      return(icon);
    }
  }
}
```

Here is one implementation of `DetailForm` that works with the revised `LunchList`:

```
package apt.tutorial;

import android.app.Activity;
import android.database.sqlite.SQLiteDatabase;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.RadioGroup;
import android.widget.TextView;

public class DetailForm extends Activity {
  EditText name=null;
  EditText address=null;
  EditText notes=null;
  RadioGroup types=null;
  Restaurant current=null;
  SQLiteDatabase db=null;
  String restaurantId=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.detail_form);
    db=(new RestaurantSQLiteHelper(this))
                                    .getWritableDatabase();
```

```
    name=(EditText)findViewById(R.id.name);
    address=(EditText)findViewById(R.id.addr);
    notes=(EditText)findViewById(R.id.notes);
    types=(RadioGroup)findViewById(R.id.types);

    Button save=(Button)findViewById(R.id.save);

    save.setOnClickListener(onSave);

    restaurantId=getIntent().getStringExtra(LunchList.ID_EXTRA);

    if (restaurantId==null) {
      current=new Restaurant();
    }
    else {
      load();
    }
  }

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

    db.close();
  }

  private void load() {
    current=Restaurant.getById(restaurantId, db);

    name.setText(current.getName());
    address.setText(current.getAddress());
    notes.setText(current.getNotes());

    if (current.getType().equals("sit_down")) {
      types.check(R.id.sit_down);
    }
    else if (current.getType().equals("take_out")) {
      types.check(R.id.take_out);
    }
    else {
      types.check(R.id.delivery);
    }
  }

  private View.OnClickListener onSave=new View.OnClickListener() {
    public void onClick(View v) {
      current.setName(name.getText().toString());
      current.setAddress(address.getText().toString());
      current.setNotes(notes.getText().toString());

      switch (types.getCheckedRadioButtonId()) {
        case R.id.sit_down:
          current.setType("sit_down");
          break;
```

```
        case R.id.take_out:
          current.setType("take_out");
          break;

        case R.id.delivery:
          current.setType("delivery");
          break;
      }

      if (restaurantId==null) {
        current.save(db);
      }
      else {
        current.update(restaurantId, db);
      }

      finish();
    }
  };
}
```

And, here is an implementation of `Restaurant` with the changes needed by `DetailForm`:

```
package apt.tutorial;

import android.content.ContentValues;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;

public class Restaurant {
  private String name="";
  private String address="";
  private String type="";
  private String notes="";

  static Cursor getAll(SQLiteDatabase db) {
    return(db.rawQuery("SELECT * FROM restaurants ORDER BY name",
                       null));
  }

  static Restaurant getById(String id, SQLiteDatabase db) {
    String[] args={id};
    Cursor c=db.rawQuery("SELECT * FROM restaurants WHERE _id=?",
                         args);

    c.moveToFirst();

    Restaurant result=new Restaurant().loadFrom(c);

    c.close();
```

```
    return(result);
  }

  Restaurant loadFrom(Cursor c) {
    name=c.getString(c.getColumnIndex("name"));
    address=c.getString(c.getColumnIndex("address"));
    type=c.getString(c.getColumnIndex("type"));
    notes=c.getString(c.getColumnIndex("notes"));

    return(this);
  }

  public String getName() {
    return(name);
  }

  public void setName(String name) {
    this.name=name;
  }

  public String getAddress() {
    return(address);
  }

  public void setAddress(String address) {
    this.address=address;
  }

  public String getType() {
    return(type);
  }

  public void setType(String type) {
    this.type=type;
  }

  public String getNotes() {
    return(notes);
  }

  public void setNotes(String notes) {
    this.notes=notes;
  }

  public String toString() {
    return(getName());
  }

  void save(SQLiteDatabase db) {
    ContentValues cv=new ContentValues();

    cv.put("name", name);
    cv.put("address", address);
```

```
    cv.put("type", type);
    cv.put("notes", notes);
    db.insert("restaurants", "name", cv);
  }

  void update(String id, SQLiteDatabase db) {
    ContentValues cv=new ContentValues();
    String[] args={id};

    cv.put("name", name);
    cv.put("address", address);
    cv.put("type", type);
    cv.put("notes", notes);
    db.update("restaurants", cv, "_id=?", args);
  }
}
```

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Have the `Restaurant` hold a URL for the restaurant's Web site. Update the UI to collect this address in the detail form. Launch that URL via `startActivity()` via an option menu choice from the restaurant list, so you can view the restaurant's Web site.
- Add an option menu to delete a `Restaurant`. Raise an `AlertDialog` to confirm that the user wants the `Restaurant` deleted. Delete it from the database and refresh the list if the user confirms the deletion.

What's Your Preference?

In this tutorial, we will add a preference setting for the sort order of the `Restaurant` list. To do this, we will create a `PreferenceScreen` definition in XML, load that into a `PreferenceActivity`, connect that activity to the application, and finally actually use the preference to control the sort order.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `12-Activities` edition of `LunchList` to use as a starting point.

Step #1: Define the Preference XML

First, add a `LunchList/res/xml/preferences.xml` file as follows:

```
<PreferenceScreen
  xmlns:android="http://schemas.android.com/apk/res/android">
  <ListPreference
    android:key="sort_order"
    android:title="Sort Order"
    android:summary="Choose the order the list uses"
    android:entries="@array/sort_names"
    android:entryValues="@array/sort_clauses"
    android:dialogTitle="Choose a sort order" />
</PreferenceScreen>
```

This sets up a single-item `PreferenceScreen`. Note that it references two string arrays, one for the display labels of the sort-order selection list, and one for the values actually stored in the `SharedPreferences`.

So, to define those string arrays, add a `LunchList/res/values/arrays.xml` file with the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string-array name="sort_names">
    <item>By Name, Ascending</item>
    <item>By Name, Descending</item>
    <item>By Type</item>
    <item>By Address, Ascending</item>
    <item>By Address, Descending</item>
  </string-array>
  <string-array name="sort_clauses">
    <item>ORDER BY name ASC</item>
    <item>ORDER BY name DESC</item>
    <item>ORDER BY type, name ASC</item>
    <item>ORDER BY address ASC</item>
    <item>ORDER BY address DESC</item>
  </string-array>
</resources>
```

Note we are saying that the value stored in the `SharedPreferences` will actually be an `ORDER BY` clause for use in our SQL query. This is a convenient trick, though it does tend to make the system a bit more fragile – if we change our column names, we might have to change our preferences to match and deal with older invalid preference values.

Step #2: Create the Preference Activity

Next, we need to create a `PreferenceActivity` that will actually use these preferences. To do this, add a `LunchList/src/apt/tutorial/EditPreferences.java` file with our `PreferenceActivity` implementation:

```
package apt.tutorial;

import android.app.Activity;
import android.os.Bundle;
import android.preference.PreferenceActivity;
```

```
public class EditPreferences extends PreferenceActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    addPreferencesFromResource(R.xml.preferences);
  }
}
```

We also need to update `AndroidManifest.xml` to reference this activity, so we can launch it later:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="apt.tutorial"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:label="@string/app_name">
        <activity android:name=".LunchList"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".DetailForm">
        </activity>
        <activity android:name=".EditPreferences">
        </activity>
    </application>
</manifest>
```

Step #3: Connect the Preference Activity to the Option Menu

Now, we can add a menu option to launch the `EditPreferences` activity.

We need to add another `<item>` to our `LunchList/res/menu/option.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/add"
    android:title="Add"
    android:icon="@android:drawable/ic_menu_add"
  />
  <item android:id="@+id/prefs"
```

```
    android:title="Settings"
    android:icon="@android:drawable/ic_menu_preferences"
  />
  <item android:id="@+id/run"
    android:title="Run Long Task"
    android:icon="@drawable/run"
  />
</menu>
```

Once again, we are using a built-in system icon for the menu choice, representing an edit-preference action.

Of course, if we modify the menu XML, we also need to modify the `LunchList` implementation of `onOptionsItemSelected()` to match, so replace the current implementation with the following:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.add) {
    startActivity(new Intent(this, DetailForm.class));

    return(true);
  }
  else if (item.getItemId()==R.id.prefs) {
    startActivity(new Intent(this, EditPreferences.class));

    return(true);
  }
  else if (item.getItemId()==R.id.run) {
    startWork();

    return(true);
  }

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

All we are doing is starting up our `EditPreferences` activity.

If you recompile and reinstall the application, you will see our new menu option:

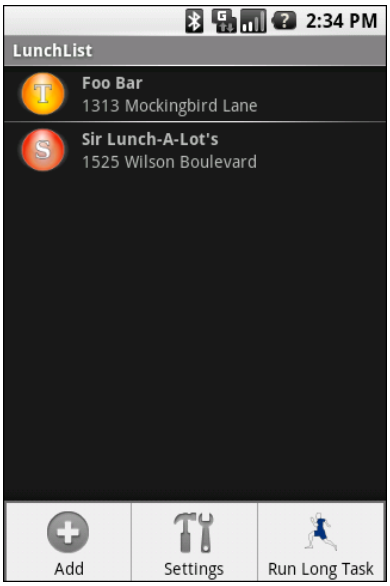


Figure 27. The LunchList with the new menu option

And if you choose that menu option, you will get the `EditPreferences` activity:

Figure 28. The preferences activity

Clicking the Sort Order item will bring up a selection list of available sort orders:

Figure 29. The available sort orders

Of course, none of this is actually having any effect on the sort order itself, which we will address in the next section.

Step #4: Apply the Sort Order on Startup

Now, given that the user has chosen a sort order, we need to actually use it. First, we can apply it when the application starts up – the next section will handle changing the sort order after the user changes the preference value.

First, the `getAll()` method on `Restaurant` needs to take a sort order as a parameter, rather than apply one of its own. So, change that method as follows:

```
static Cursor getAll(SQLiteDatabase db, String orderBy) {
  return(db.rawQuery("SELECT * FROM restaurants "+orderBy,
                     null));
}
```

Then, we need to get our hands on our `SharedPreferences` instance. Add imports to `LunchList` for `android.content.SharedPreferences` and `android.preference.PreferenceManager`, along with a `SharedPreferences` data member named `prefs`.

Next, add this line to the end of `onCreate()` in `LunchList`, to initialize `prefs` to be the `SharedPreferences` our preference activity uses:

```
prefs=PreferenceManager.getDefaultSharedPreferences(this);
```

Finally, change the call to `getAll()` to use the `SharedPreferences`:

```
model=Restaurant.getAll(db, prefs.getString("sort_order", ""));
```

Here, we use an empty string as the default value, so if the user has not specified a sort order yet, the sort order will be "natural" (i.e., whatever SQLite feels like returning).

Now, if you recompile and reinstall the application, then set a sort order preference, you can see that preference take effect if you exit and reopen the application.

Step #5: Listen for Preference Changes

That works, but users will get annoyed if they have to exit the application just to get their preference choice to take effect. To change the sort order on the fly, we first need to know when they change the sort order.

`SharedPreferences` has the notion of a preference listener object, to be notified on such changes. To take advantage of this, add the following line at the end of `onCreate()` in `LunchList`:

```
prefs.registerOnSharedPreferenceChangeListener(prefListener);
```

This snippet refers to a `prefListener` object, so add the following code to `LunchList` to create a stub implementation of that object:

```
private SharedPreferences.OnSharedPreferenceChangeListener prefListener=
  new SharedPreferences.OnSharedPreferenceChangeListener() {
  public void onSharedPreferenceChanged(SharedPreferences sharedPrefs, String
key) {
    if (key.equals("sort_order")) {
    }
  }
};
```

All we are doing right now is watching for our specific preference of interest (`sort_order`), though we are not actually taking advantage of the changed value.

Step #6: Re-Apply the Sort Order on Changes

Finally, we actually need to change the sort order. For simple lists like this, the easiest way to accomplish this is to get a fresh `Cursor` representing our list (from `getAll()` on `Restaurant`) with the proper sort order, and use the new `Cursor` instead of the old one.

First, rather than have our `ListView` be a local variable in `onCreate()`, make it be a data member of `LunchList`. Be sure to change `onCreate()` to get rid of the local variable, though.

Then, pull some of the list-population logic out of `onCreate()`, by implementing an `initList()` method as follows:

```
private void initList() {
  if (model!=null) {
    stopManagingCursor(model);
    model.close();
  }

  model=Restaurant.getAll(db, prefs.getString("sort_order", ""));
  startManagingCursor(model);

  adapter=new RestaurantAdapter(model);
  list.setAdapter(adapter);
}
```

Note that we call `stopManagingCursor()` so Android will ignore the old `Cursor`, then we close it, before we get and apply the new `Cursor`. Of course, we only do those things if there is an old `Cursor`.

The `onCreate()` method needs to change to take advantage of `initList()`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);
  progress=(ProgressBar)findViewById(R.id.progress);

  db=(new RestaurantSQLiteHelper(this))
                                  .getWritableDatabase();

  list=(ListView)findViewById(R.id.restaurants);
  prefs=PreferenceManager.getDefaultSharedPreferences(this);
  list.setOnItemClickListener(onListClick);
  initList();

  prefs.registerOnSharedPreferenceChangeListener(prefListener);
}
```

Also, we can call `initList()` from `prefListener`. Since we cannot be sure what thread we will be called on for our listener callback method, it is safest to wrap this call to `initList()` in `runOnUiThread()`. Our implementation of `prefListener` should now look like:

```
private SharedPreferences.OnSharedPreferenceChangeListener prefListener=
  new SharedPreferences.OnSharedPreferenceChangeListener() {
  public void onSharedPreferenceChanged(SharedPreferences sharedPrefs, String
key) {
    if (key.equals("sort_order")) {
      runOnUiThread(new Runnable() {
        public void run() {
          initList();
        }
      });
    }
  }
};
```

At this point, if you recompile and reinstall the application, you should see the sort order change immediately as you change the order via the preferences.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Add a preference for the default type of `Restaurant` (e.g., take-out). Use that preference in detail forms when creating a new `Restaurant`.
- Add an option menu to the detail form activity and have it be able to start the preference activity the way we did from the option menu for the list.
- Rather than use preferences, store the preference values in a JSON file that you read in at startup and re-read in `onResume()` (to find out about changes). This means you will need to create your own preference UI, rather than rely upon the one created by the preference XML.

Turn, Turn, Turn

In this tutorial, we will make our application somewhat more intelligent about screen rotations, ensuring that partially-entered `Restaurant` information remains intact even after the screen rotates.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `13-Prefs` edition of `LunchList` to use as a starting point.

Step #1: Add a Stub onSaveInstanceState() Implementation

Since we are not holding onto network connections or other things that cannot be stored in a `Bundle`, we can use `onSaveInstanceState()` to track our state as the screen is rotated.

To that end, add a stub implementation of `onSaveInstanceState()` to `DetailForm` as follows:

```
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
```

```
  super.onSaveInstanceState(savedInstanceState);
}
```

Step #2: Pour the Form Into the Bundle

Now, fill in the details of `onSaveInstanceState()`, putting our widget contents into the supplied `Bundle`:

```
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
  super.onSaveInstanceState(savedInstanceState);

  savedInstanceState.putString("name",
                               name.getText().toString());
  savedInstanceState.putString("address",
                               address.getText().toString());
  savedInstanceState.putString("notes",
                               notes.getText().toString());
  savedInstanceState.putInt("type",
                            types.getCheckedRadioButtonId());
}
```

Step #3: Detect a Restart and Repopulate the Form

Next, we need to determine if our activity was restarted. To do this, we can just look to see if the `Bundle` passed into `onCreate()` is `null`. A non-`null` `Bundle` means the activity is being restarted, such as after a screen rotation.

Of course, if this is a restart, we need to fill in the widget contents from the `Bundle`. More importantly, we need to do this last, so any changes to the form from existing logic (e.g., loading a `Restaurant` out of the database for editing) get overwritten with our `Bundle` contents.

Here is the code you should add to the end of `onCreate()` to accomplish these tasks:

```
if (savedInstanceState!=null) {
  name.setText(savedInstanceState.getString("name"));
  address.setText(savedInstanceState.getString("address"));
  notes.setText(savedInstanceState.getString("notes"));
  types.check(savedInstanceState.getInt("type"));
}
```

At this point, you can recompile and reinstall the application. Use <Ctrl>-<F12> to simulate rotating the screen of your emulator. If you do this after making changes (but not saving) on the DetailForm, you will see those changes survive the rotation.

Step #4: Fix Up the Landscape Detail Form

As you tested the work from the previous section, you no doubt noticed that the DetailForm layout is not well-suited for landscape – the notes text area is chopped off and the Save button is missing. To fix this, we need to create a LunchList/res/layout-land/detail_form.xml file, derived from our original, but set up to take advantage of the whitespace to the right of the radio buttons:

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:stretchColumns="1,3"
  >
  <TableRow>
    <TextView android:text="Name:" />
    <EditText android:id="@+id/name"
      android:layout_span="3"
    />
  </TableRow>
  <TableRow>
    <TextView android:text="Address:" />
    <EditText android:id="@+id/addr"
      android:layout_span="3"
    />
  </TableRow>
  <TableRow>
    <TextView android:text="Type:" />
    <RadioGroup android:id="@+id/types">
      <RadioButton android:id="@+id/take_out"
        android:text="Take-Out"
      />
      <RadioButton android:id="@+id/sit_down"
        android:text="Sit-Down"
      />
      <RadioButton android:id="@+id/delivery"
        android:text="Delivery"
      />
    </RadioGroup>
    <TextView android:text="Notes:" />
    <LinearLayout
      android:layout_width="fill_parent"
```

```
      android:layout_height="fill_parent"
      android:orientation="vertical"
      >
      <EditText android:id="@+id/notes"
        android:singleLine="false"
        android:gravity="top"
        android:lines="4"
        android:scrollHorizontally="false"
        android:maxLines="4"
        android:maxWidth="140sp"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
      />
      <Button android:id="@+id/save"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Save"
      />
    </LinearLayout>
  </TableRow>
</TableLayout>
```

Now, if you recompile and reinstall the application, you should see a better landscape rendition of the detail form:

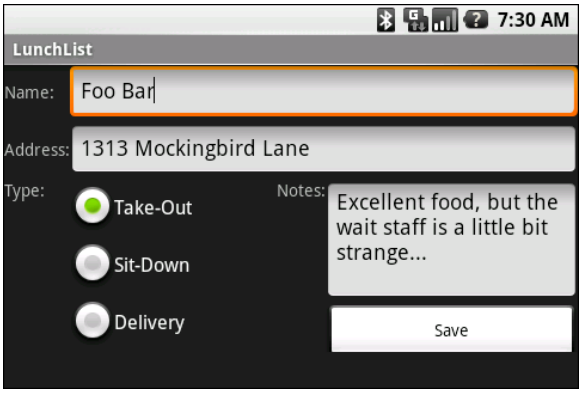


Figure 30. The new landscape layout

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Try switching to `onRetainConfigurationInstance()` instead of `onSaveInstanceState()`.

- Try commenting out `onSaveInstanceState()`. Does the activity still retain its instance state? Why or why not?
- Have the application automatically rotate based on physical orientation instead of keyboard position. Hint: find a place to apply `android:screenOrientation = "sensor"`.

Asking Permission

In this tutorial, we will add a bit of code that asks permission to place a call, and we will add a phone number to our `Restaurant` data model and detail form. In the next tutorial, we will actually place the call.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `14-Rotation` edition of `LunchList` to use as a starting point.

Step #1: Add a Phone Number to the Database Schema

If we want our phone numbers to stick around, we need to put them in the database.

With that in mind, update `RestaurantSQLiteHelper` to use the following implementation of `onCreate()`:

```
@Override
public void onCreate(SQLiteDatabase db) {
  Cursor c=db.rawQuery("SELECT name FROM sqlite_master WHERE type='table' AND
name='restaurants'", null);
```

```
  try {
    if (c.getCount()==0) {
      db.execSQL("CREATE TABLE restaurants "+
                 "(_id INTEGER PRIMARY KEY AUTOINCREMENT,"+
                 "name TEXT, address TEXT, type TEXT,"+
                 "notes TEXT, phoneNumber TEXT);");
    }
  }
  finally {
    c.close();
  }
}
```

Any time you make a material modification to the schema, you also need to increment the schema version number. For `RestaurantSQLiteHelper`, that is held in `SCHEMA_VERSION`, so increment to `2`:

```
private static final int SCHEMA_VERSION=2;
```

Step #2: Intelligently Handle Database Updates

When the schema version increments, `onUpgrade()` is called on `RestaurantSQLiteHelper` rather than `onCreate()`. Our original version of `onUpgrade()` simply dropped the `restaurants` table and recreated it. While that works, it dumps all of our data.

To hang onto that data, we need to copy the current data to a temporary location, then update the schema for `restaurants`, then restore the data back into the `restaurants` table. To do this, replace the current `onUpgrade()` with the following:

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
  android.util.Log.w("LunchList", "Upgrading database, which will destroy all
old data");

  if (oldVersion==1 && newVersion==2) {
    db.execSQL("CREATE TEMP TABLE r "+
               "(_id INTEGER PRIMARY KEY AUTOINCREMENT,"+
               "name TEXT, address TEXT, type TEXT,"+
               "notes TEXT);");
    db.execSQL("INSERT INTO r SELECT _id, name, address, type, notes FROM
restaurants");
  }
```

```
  db.execSQL("DROP TABLE IF EXISTS restaurants");
  onCreate(db);

  if (oldVersion==1 && newVersion==2) {
    db.execSQL("INSERT INTO restaurants SELECT _id, name, address, type, notes,
NULL FROM r");
  }
}
```

Step #3: Add a Phone Number to the Restaurant Model

With the database schema in place, we need to update `Restaurant` to work with the new `phoneNumber` column. That is simply a matter of cloning every statement referencing, say, `notes`, and making the clone reference `phoneNumber`, such as:

```
private String notes="";
private String phoneNumber="";
```

and:

```
notes=c.getString(c.getColumnIndex("notes"));
phoneNumber=c.getString(c.getColumnIndex("phoneNumber"));
```

Make these changes throughout `Restaurant`. Be sure to add a `getPhoneNumber()` and `setPhoneNumber()` accessor pair, so `DetailForm` can get and set the phone number.

Step #4: Collect the Phone Number on the Detail Form

If we actually want to have phone numbers, though, we need to actually collect them on `DetailForm`.

First, update `LunchList/res/layout/detail_form.xml` to add the following after the address row in our `TableLayout`:

```
<TableRow>
  <TextView android:text="Phone:" />
  <EditText android:id="@+id/phone"
    android:phoneNumber="true"
  />
</TableRow>
```

Similarly, add the following after the address row in `LunchList/res/layout-land/detail_form.xml`:

```
<TableRow>
  <TextView android:text="Phone:" />
  <EditText android:id="@+id/phone"
    android:phoneNumber="true"
    android:layout_span="3"
  />
</TableRow>
```

Then, as in the previous section, clone all references to notes in `DetailForm` to make references to our `phone` widgets the corresponding methods on `Restaurant`, such as:

```
EditText notes=null;
EditText phone=null;
```

and:

```
notes=(EditText)findViewById(R.id.notes);
phone=(EditText)findViewById(R.id.phone);
```

At this point, you can recompile and reinstall the application. When you first run it, there will be a tiny pause as the database is updated. After that point, you can use the new field to add phone numbers to whichever `Restaurant` object(s) you want:

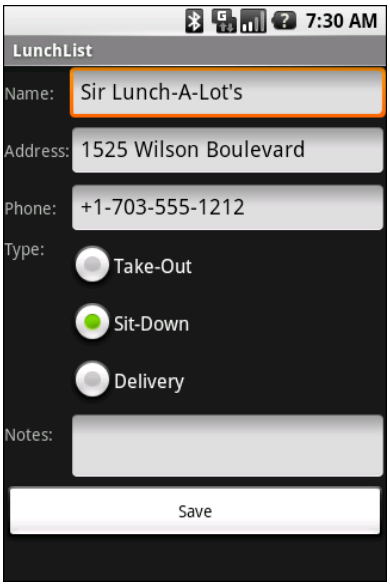


Figure 31. The new DetailForm layout

Step #5: Ask for Permission to Make Calls

Finally, we can update `AndroidManifest.xml` to put in a permission request to be able to place phone calls:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="apt.tutorial"
      android:versionCode="1"
      android:versionName="1.0">
    <uses-permission android:name="android.permission.CALL_PHONE"/>
    <application android:label="@string/app_name">
        <activity android:name=".LunchList"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".DetailForm">
        </activity>
        <activity android:name=".EditPreferences">
        </activity>
    </application>
</manifest>
```

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Add a preference to display the phone number instead of the address in the `Restaurant` list. Have the list detect the preference and fill in the second line of the `Restaurant` rows accordingly.
- Push your APK file to a Web site that is configured to support the proper MIME type for Android application downloads (e.g., Amazon S3). Try installing your APK onto a device from the published location, to see how your requested permission appears to end users at install time.
- Find a revision to the `layout-land` version of `detail_form.xml` that does not clip the bottom radio button.

Calling For a Search

In this tutorial, we will place a call based on the phone number entered for our `Restaurant`. Also, we will enable searching of the `Restaurant` list, so you can find one in the vast array of `Restaurant` objects you surely will maintain in this application.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `15-Perms` edition of `LunchList` to use as a starting point.

Step #1: Dial the Number

First, let us set up `DetailForm` with its own option menu that contains a Call item. When chosen, we dial the phone number, assuming there is one.

First, create `LunchList/res/menu/option_detail.xml` with the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/call"
    android:title="Call"
    android:icon="@android:drawable/ic_menu_call"
```

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

Then, add the following methods to `DetailForm`:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  new MenuInflater(getApplication())
                                .inflate(R.menu.option_detail, menu);

  return(super.onCreateOptionsMenu(menu));
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.call) {
    if (current.getPhoneNumber()!=null &&
        current.getPhoneNumber().length()>0) {
      String toDial="tel:"+current.getPhoneNumber();

      startActivity(new Intent(Intent.ACTION_DIAL,
                               Uri.parse(toDial)));
    }
  }

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

Note that you will need to add a number of imports (`Intent`, `Menu`, `MenuInflater`, `MenuItem`, and `Uri`) to get this to compile cleanly.

In the new code, we check to see if there is a phone number. If so, we wrap the phone number in a `tel:` `Uri`, then put that in an `ACTION_DIAL` `Intent` and start an activity on that `Intent`. This puts the phone number in the dialer.

If you rebuild and reinstall the application and try out the new menu choice on some `Restaurant` with a phone number, you will see the `Dialer` appear:

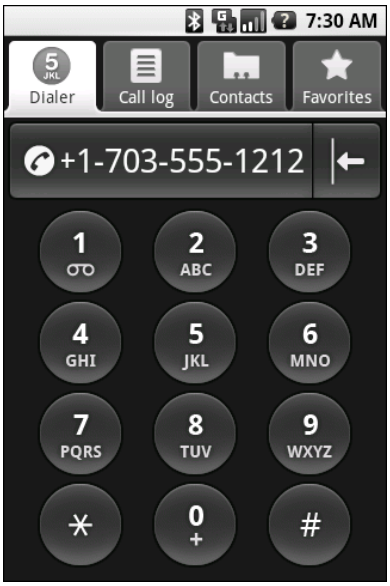


Figure 32. The Dialer

Step #2: Make the Call

Suppose we want to take advantage of the `CALL_PHONE` permission we requested in the previous tutorial. All that we need to do is switch our `Intent` from `ACTION_DIAL` to `ACTION_CALL`:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.call) {
    if (current.getPhoneNumber()!=null &&
        current.getPhoneNumber().length()>0) {
      String toDial="tel:"+current.getPhoneNumber();

      startActivity(new Intent(Intent.ACTION_CALL,
                               Uri.parse(toDial)));
    }
  }

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

Now, if you rebuild and reinstall the application, and try choosing the Call option menu item, you will immediately "call" the phone number...which

will actually place a phone call if you are trying this on a device. The emulator, of course, cannot place phone calls.

Step #3: Find Matching Restaurants

In order to search the list of `Restaurant` objects, we need to be able to specify a `WHERE` clause when calling `getAll()` on the `Restaurant` class. Otherwise, it will always return all `Restaurant` objects, whether we want them or not.

So, modify `getAll()` on `Restaurant` to look like the following:

```
static Cursor getAll(SQLiteDatabase db, String where,
                     String orderBy) {
  StringBuffer query=new StringBuffer("SELECT * FROM restaurants ");

  if (where!=null) {
    query.append("WHERE ");
    query.append(where);
  }

  query.append(orderBy);

  return(db.rawQuery(query.toString(), null));
}
```

Step #4: Have the List Conduct the Search

First, we need to be able to trigger the local search. To do this, add another item to our `LunchList/res/menu/options.xml` menu definition:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/add"
    android:title="Add"
    android:icon="@android:drawable/ic_menu_add"
  />
  <item android:id="@+id/search"
    android:title="Search"
    android:icon="@android:drawable/ic_menu_search"
  />
  <item android:id="@+id/prefs"
    android:title="Settings"
    android:icon="@android:drawable/ic_menu_preferences"
  />
```

```
  <item android:id="@+id/run"
    android:title="Run Long Task"
    android:icon="@drawable/run"
  />
</menu>
```

Then, add the corresponding segment to `onOptionsItemSelected()` in `LunchList` to handle our new item:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.add) {
    startActivity(new Intent(this, DetailForm.class));

    return(true);
  }
  else if (item.getItemId()==R.id.prefs) {
    startActivity(new Intent(this, EditPreferences.class));

    return(true);
  }
  else if (item.getItemId()==R.id.search) {
    onSearchRequested();
    return(true);
  }
  else if (item.getItemId()==R.id.run) {
    startWork();
```

Finally, in `initList()` in `LunchList`, we need to see if our activity was launched via a search and, if so, use the search string entered by the user. Here is a revised `initList()` implementation for `LunchList` that does just that:

```
private void initList() {
  if (model!=null) {
    stopManagingCursor(model);
    model.close();
  }

  String where=null;

  if (Intent.ACTION_SEARCH.equals(getIntent().getAction())) {
    where="name LIKE \"%"+getIntent().getStringExtra(SearchManager.QUERY)+"%\"";
  }

  model=Restaurant.getAll(db, where, prefs.getString("sort_order", ""));
  startManagingCursor(model);

  adapter=new RestaurantAdapter(model);
```

```
  list.setAdapter(adapter);
}
```

Step #5: Integrate the Search in the Application

Finally, we need to tell Android that this application is searchable and how to do the search.

First, we need to overhaul our `AndroidManifest.xml` file to indicate that the `LunchList` activity is both searchable and the activity to launch to conduct a search:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="apt.tutorial"
      android:versionCode="1"
      android:versionName="1.0">
    <uses-permission android:name="android.permission.CALL_PHONE"/>
    <application android:label="@string/app_name">
        <activity android:name=".LunchList"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
            <intent-filter>
              <action android:name="android.intent.action.SEARCH" />
              <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
            <meta-data android:name="android.app.searchable"
                  android:resource="@xml/searchable" />
            <meta-data android:name="android.app.default_searchable"
                android:value=".LunchList" />
        </activity>
        <activity android:name=".DetailForm">
        </activity>
        <activity android:name=".EditPreferences">
        </activity>
    </application>
</manifest>
```

The `android.app.searchable` metadata element names `@xml/searchable` as being the configuration details for the search itself. So, add a `LunchList/res/xml/searchable.xml` file with the following content:

```
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
    android:label="@string/searchLabel"
    android:hint="@string/searchHint" />
```

That file, in turn, references a pair of `String` resources, so we need to add them to `LunchList/res/values/strings.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">LunchList</string>
    <string name="searchLabel">Restaurants</string>
    <string name="searchHint">lorem</string>
</resources>
```

At this point, you can recompile and reinstall your application. Choosing the option menu will display our new search item:

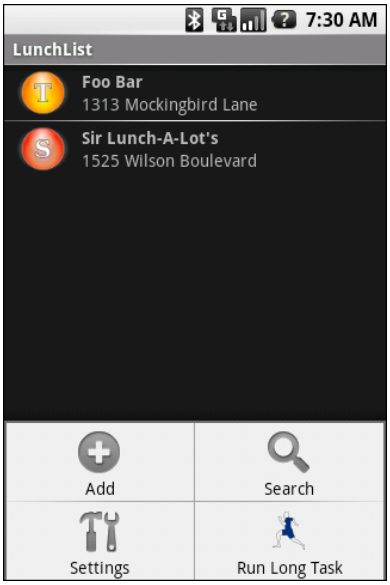


Figure 33. The new option menu

Choosing the search brings up the search field:

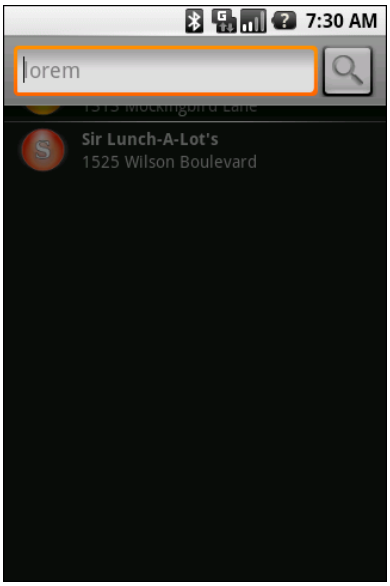


Figure 34. The search field

Entering in some value will give us a new `LunchList` with just the matching subset.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Add a preference so users can determine whether they want the number to be dialed (put in the `Dialer`) or directly called. Make use of this preference to determine whether to dial or call the number.
- Extend the search to search all relevant attributes of the data model, including address, notes, and the phone number.

PART II – Network Application Tutorials

Raising a Tweet

In this tutorial, we will experiment with the Apache HttpClient library in Android, using it to post a status update to our Twitter account. Twitter clients are the "Hello, world!" applications of the Web 2.0, REST API era, so over the course of several tutorials, we will focus on Twitter-related functionality in an application we will call `Patchy` (short for "Patchy: Another Twitter Client? Heck, Yeah!").

Step-By-Step Instructions

This tutorial starts a new application, independent from the `LunchList` application developed in the preceding tutorials. Hence, we will have you create a new application from scratch.

Step #1: Set Up a Twitter Account

Twitter is a micro-blogging service that you may have heard about.

If you have a Twitter account and do not mind using it for creating a `Patchy` application, feel free to stick with it. Otherwise, create a Twitter account that you can use for your experimentation. Visit http://twitter.com and sign up.

If you are creating an account solely for this class, or a disposable one for `Patchy` experimentation, please be sure to delete the account when you are done. You can delete your Twitter account via a link from your Settings page.

Step #2: Create a Stub Application and Activity

Using Eclipse or `activitycreator`, make a project named `Patchy` with a stub activity named `apt.tutorial.two.Patchy`. The generated activity class should resemble the following:

```
package apt.tutorial.two;

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

public class Patchy extends Activity
{
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

Step #3: Create a Layout

Next, we need a layout for use with the `Patchy` activity. At the outset, we want:

- A field for your Twitter user name
- A field for your Twitter password, needed to use much of the Twitter REST API
- A multi-line field for a status update you want to publish
- A Send button that will send the status update

Here is a simple `TableLayout` containing those items, one that looks a bit like the `DetailForm` from the `LunchList` tutorial earlier in this book:

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:stretchColumns="1"
  >
  <TableRow>
    <TextView android:text="User Name:" />
    <EditText android:id="@+id/user" />
  </TableRow>
  <TableRow>
    <TextView android:text="Password:" />
    <EditText android:id="@+id/password"
      android:password="true"
    />
  </TableRow>
  <TableRow>
    <TextView android:text="Status:" />
    <EditText android:id="@+id/status"
      android:singleLine="false"
      android:gravity="top"
      android:lines="5"
      android:scrollHorizontally="false"
      android:maxLines="5"
      android:maxWidth="200sp"
    />
  </TableRow>
  <Button android:id="@+id/send"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Send"
  />
</TableLayout>
```

If you compile and reinstall this application, you will see the layout in action:

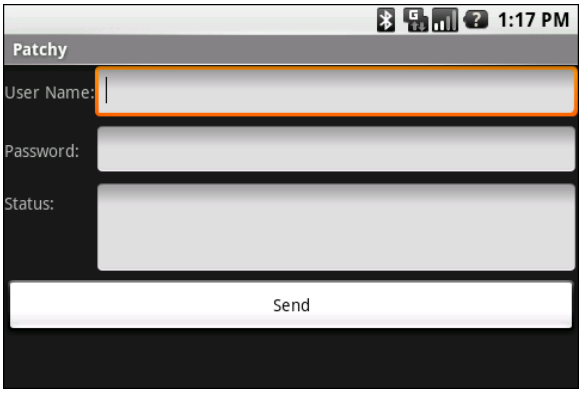


Figure 35. The initial Patchy layout, in landscape

Step #4: Listen for Send Actions

Next, we need to get control when the user clicks the Send button. This involves finding the Send button in our layout and attaching a listener to it.

Replace the stock implementation of `Patchy` with the following:

```
package apt.tutorial.two;

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

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

    Button send=(Button)findViewById(R.id.send);

    send.setOnClickListener(onSend);
  }

  private View.OnClickListener onSend=new View.OnClickListener() {
    public void onClick(View v) {
    }
  };
}
```

Step #5: Make the Twitter Request

With all that done, we can actually invoke the Twitter REST API. This is somewhat tedious, for a few reasons:

- Twitter requires HTTP Basic authentication to pass over the user name and password, rather than embedding them as values in, say, an HTTP POST request. HttpClient does not handle preemptive HTTP authentication very well.
- HTTP Basic authentication requires Base64 encoding, and Android lacks a publicly-visible Base64 encoder.

- Twitter requires an `Expect: 100-Continue` HTTP header, so it can validate the authentication before receiving the actual HTTP POST body as a whole. This is possible with HttpClient, but it is not automatic.

Here is what you need to do to get past all of this and make `Patchy` work:

First, find a suitable Base64 encoder, such as this public domain one. Put the `Base64` class in your project in `Patchy/src/apt/tutorial/two/` and add the matching package line to the top of the file, so it is available to your `Patchy` implementation.

Next, replace your existing `Patchy` implementation with the following:

```
package apt.tutorial.two;

import android.app.Activity;
import android.app.AlertDialog;
import android.os.Bundle;
import android.view.View;
import android.util.Log;
import android.widget.Button;
import android.widget.EditText;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import org.apache.http.NameValuePair;
import org.apache.http.HttpVersion;
import org.apache.http.client.ResponseHandler;
import org.apache.http.client.HttpClient;
import org.apache.http.client.entity.UrlEncodedFormEntity;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.impl.client.BasicResponseHandler;
import org.apache.http.impl.client.DefaultHttpClient;
import org.apache.http.message.BasicNameValuePair;
import org.apache.http.params.BasicHttpParams;
import org.apache.http.params.HttpParams;
import org.apache.http.params.HttpProtocolParams;
import org.apache.http.protocol.HTTP;
import org.json.JSONObject;

public class Patchy extends Activity {
  private DefaultHttpClient client=null;
  private EditText user=null;
  private EditText password=null;
  private EditText status=null;
```

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

  user=(EditText)findViewById(R.id.user);
  password=(EditText)findViewById(R.id.password);
  status=(EditText)findViewById(R.id.status);

  Button send=(Button)findViewById(R.id.send);

  send.setOnClickListener(onSend);

  client=new DefaultHttpClient();

  HttpParams params=new BasicHttpParams();

  HttpProtocolParams.setVersion(params, HttpVersion.HTTP_1_1);
  HttpProtocolParams.setContentCharset(params, "UTF_8");
  HttpProtocolParams.setUseExpectContinue(params, false);

  client.setParams(params);
}

private String getCredentials() {
  String u=user.getText().toString();
  String p=password.getText().toString();

  return(Base64.encodeBytes((u+":"+p).getBytes()));
}

private void updateStatus() {
  try {
    String s=status.getText().toString();

    HttpPost post=new HttpPost("http://twitter.com/statuses/update.json");

    post.addHeader("Authorization",
                   "Basic "+getCredentials());

    List<NameValuePair> form=new ArrayList<NameValuePair>();

    form.add(new BasicNameValuePair("status", s));

    post.setEntity(new UrlEncodedFormEntity(form, HTTP.UTF_8));

    ResponseHandler<String> responseHandler=new BasicResponseHandler();
    String responseBody=client.execute(post, responseHandler);
    JSONObject response=new JSONObject(responseBody);
  }
  catch (Throwable t) {
    Log.e("Patchy", "Exception in updateStatus()", t);
    goBlooey(t);
  }
```

```
  }

  private void goBlooey(Throwable t) {
    AlertDialog.Builder builder=new AlertDialog.Builder(this);

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

  private View.OnClickListener onSend=new View.OnClickListener() {
    public void onClick(View v) {
      updateStatus();
    }
  };
}
```

Here, we:

- Set up an `DefaultHttpClient` instance for accessing the Apache HttpClient engine, complete with support for the `Expect: 100-Continue` HTTP header
- Get access to our `EditText` widgets from the layout
- On a Send button click, call `updateStatus()`
- In `updateStatus()`, we create an `HttpPost` object to represent the request, fill in the authentication credentials, fill in the status as a form element, turn the whole thing into a valid HTTP POST operation, execute it, and parse the response as a JSON object

If things work, at present, we do nothing to update the activity; if an `Exception` is raised, we log it to the Android log and raise an `AlertDialog`.

If you recompile and reinstall the application, you should now be able to update your Twitter status by filling in the fields and clicking the Send button.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Process the Twitter request in a background thread and display a `Toast` when the request is complete and successful.
- Add logic to check for the Twitter message 140-character limit before sending the request. If the message is too long, display an `AlertDialog`.
- Make the `AlertDialog` from the previous point obsolete by adding the `android:maxLength` attribute to the `EditText` for the message, to constrain it to 140 characters.
- Add in TinyURL support by adding a field for a URL to append to the end of the message, then using the TinyURL REST API (`http://tinyurl.com/api-create.php?url = ...` returns the shortened URL as a response) to generate the shortened URL, then attach it to the message before making the request.

Opening a JAR

Writing our own Twitter API seems silly, considering that there are so many of them available as open source, including three for Java. In this tutorial, we will replace our HttpClient with JTwitter, an LGPL Twitter API that works with Android.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `17-HttpClient` edition of `Patchy` to use as a starting point.

Step #1: Obtain the JTwitter JAR

The official JTwitter distribution includes, in its JAR, the `org.json` package, which Android has already. It is safest to remove the `org.json` files from JTwitter before using it in Android, so there is no question of which implementation is used – besides, it will make the JAR file smaller.

Download the JTwitter JAR and, using your favorite ZIP management tool, delete the `META-INF/` and `org/` directories from the root of the JAR. Then, put the JAR in the `Patchy/libs/` directory, so it is available to your application. Or, obtain the modified JTwitter JAR from the tutorial results.

Step #2: Switch from HttpClient to JTwitter

Now, we can get rid of most of our previous `Patchy` implementation, including all of the HttpClient code, and replace it with the delightfully simply JTwitter API.

Replace the current implementation of `Patchy` with the following:

```
package apt.tutorial.two;

import android.app.Activity;
import android.app.AlertDialog;
import android.os.Bundle;
import android.view.View;
import android.util.Log;
import android.widget.Button;
import android.widget.EditText;
import winterwell.jtwitter.Twitter;

public class Patchy extends Activity {
  private EditText user=null;
  private EditText password=null;
  private EditText status=null;

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

    user=(EditText)findViewById(R.id.user);
    password=(EditText)findViewById(R.id.password);
    status=(EditText)findViewById(R.id.status);

    Button send=(Button)findViewById(R.id.send);

    send.setOnClickListener(onSend);
  }

  private void updateStatus() {
    try {
      Twitter client=new Twitter(user.getText().toString(),
                                 password.getText().toString());

      client.updateStatus(status.getText().toString());
    }
    catch (Throwable t) {
      Log.e("Patchy", "Exception in updateStatus()", t);
      goBlooey(t);
    }
  }
```

```
  private void goBlooey(Throwable t) {
    AlertDialog.Builder builder=new AlertDialog.Builder(this);

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

  private View.OnClickListener onSend=new View.OnClickListener() {
    public void onClick(View v) {
      updateStatus();
    }
  };
}
```

Besides getting rid of all HttpClient references, we reimplemented `updateStatus()` to create a `Twitter` object (with our user name and password), then called its own `updateStatus()` method with the typed-in status.

If you rebuild and reinstall `Patchy`, you can once again update your Twitter status, just with half of the lines of code.

Step #3: Create Preferences for Account Information

Rather than have the user enter their account information every time they run `Patchy`, we should store that information in user preferences, so it can be entered only on occasion.

First, we need a preference XML file, so create a new file, `Patchy/res/xml/preferences.xml`, with the following contents:

```
<PreferenceScreen
  xmlns:android="http://schemas.android.com/apk/res/android">
  <PreferenceCategory android:title="User Account">
    <EditTextPreference
      android:key="user"
      android:title="User Name"
      android:summary="Your Twitter screen name"
      android:dialogTitle="Enter your Twitter user name" />
    <EditTextPreference
```

```
      android:key="password"
      android:title="Password"
      android:summary="Your Twitter account password"
      android:password="true"
      android:dialogTitle="Enter your Twitter password" />
  </PreferenceCategory>
</PreferenceScreen>
```

Note the use of `android:password = "true"` to turn the second one into a password-style preference. Any `EditText` attributes found in the `EditTextPreference` element are passed through to the `EditText` used in the popup dialog.

Then, we need another `EditPreference` activity akin to the one we used in the `LunchList` application – call it `Patchy/src/apt/tutorial/two/EditPreferences.java`:

```
package apt.tutorial.two;

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

public class EditPreferences extends PreferenceActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    addPreferencesFromResource(R.xml.preferences);
  }
}
```

We cannot forget to update our `AndroidManifest.xml` file to reference the new activity, so amend yours to look like:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="apt.tutorial.two"
      android:versionCode="1"
      android:versionName="1.0">
  <uses-permission android:name="android.permission.INTERNET" />
    <application android:label="@string/app_name">
        <activity android:name=".Patchy"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
```

```
            </intent-filter>
        </activity>
        <activity android:name=".EditPreferences">
        </activity>
    </application>
</manifest>
```

Then, we want to have an option menu to trigger editing the preferences, so create `Patchy/res/menu/option.xml` with the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/prefs"
    android:title="Settings"
    android:icon="@android:drawable/ic_menu_preferences"
  />
</menu>
```

Finally, we need to clone the code from `LunchList` that opens our option menu using the above XML and routes the item click to the `EditPreference` activity. Paste the following methods into `Patchy`:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  new MenuInflater(getApplication())
                                .inflate(R.menu.option, menu);

  return(super.onCreateOptionsMenu(menu));
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.prefs) {
    startActivity(new Intent(this, EditPreferences.class));

    return(true);
  }

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

You will also need to add some imports (`Intent`, `Menu`, `MenuInflater`, `MenuItem`) to allow this to compile.

At this point, you can recompile and reinstall the application and see our preference activity in action, even though the values are not being used yet:

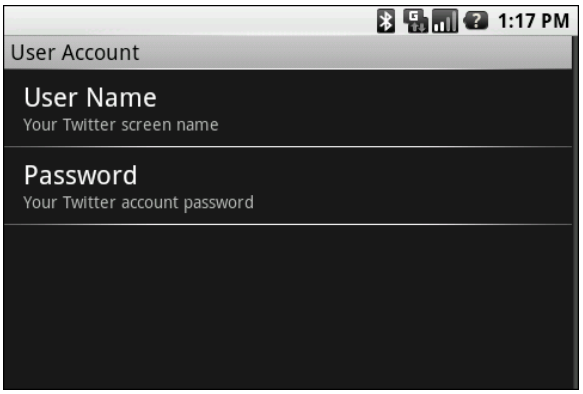


Figure 36. The preference screen for Patchy

Step #4: Use Account Information from Preferences

Now, we can get rid of the extraneous widgets on our layout and use the preferences for our account information. We can even arrange to update our `Twitter` object when the user changes account information.

First, go into `Patchy/res/layout/main.xml` and get rid of the first two `TableRow` elements, leaving you with:

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:stretchColumns="1"
  >
  <TableRow>
    <TextView android:text="Status:" />
    <EditText android:id="@+id/status"
      android:singleLine="false"
      android:gravity="top"
      android:lines="5"
      android:scrollHorizontally="false"
      android:maxLines="5"
      android:maxWidth="200sp"
    />
  </TableRow>
  <Button android:id="@+id/send"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Send"
  />
</TableLayout>
```

Then, eliminate references to the removed widgets from the list of data members and from `onCreate()`.

Next, create a private `SharedPreferences` data member named `prefs` and set it up in `onCreate()`:

```
prefs=PreferenceManager.getDefaultSharedPreferences(this);
prefs.registerOnSharedPreferenceChangeListener(prefListener);
```

The aforementioned code references an `prefListener` object, so define it as follows:

```
private SharedPreferences.OnSharedPreferenceChangeListener prefListener=
  new SharedPreferences.OnSharedPreferenceChangeListener() {
  public void onSharedPreferenceChanged(SharedPreferences sharedPrefs, String
key) {
    if (key.equals("user") || key.equals("password")) {
      resetClient();
    }
  }
};
```

Basically, when the preference changes, we want to reset our `Twitter` client. This, of course, assumes we have a `Twitter` client hanging around, so create a private `Twitter` data member named `client`, and add two methods to work with it as follows:

```
synchronized private Twitter getClient() {
  if (client==null) {
    client=new Twitter(prefs.getString("user", ""),
                       prefs.getString("password", ""));
  }

  return(client);
}

synchronized private void resetClient() {
  client=null;
}
```

Finally, in `updateStatus()`, get rid of the references to the former widgets and use `getClient()` to lazy-create our `Twitter` object, as follows:

```
private void updateStatus() {
  try {
    getClient().updateStatus(status.getText().toString());
  }
  catch (Throwable t) {
    Log.e("Patchy", "Exception in updateStatus()", t);
    goBlooey(t);
  }
}
```

The net is that we will use one `Twitter` object, on our first status update, until the application is closed or the user changes credentials.

If you rebuild and reinstall the application, you should now be able to update your Twitter status, yet only enter the status information, using the user name and password from the preferences.

Here is the complete implementation of Patchy after completing this tutorial:

```
package apt.tutorial.two;

import android.app.Activity;
import android.app.AlertDialog;
import android.content.Intent;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.preference.PreferenceManager;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.util.Log;
import android.widget.Button;
import android.widget.EditText;
import winterwell.jtwitter.Twitter;

public class Patchy extends Activity {
  private EditText status=null;
  private SharedPreferences prefs=null;
  private Twitter client=null;

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

    status=(EditText)findViewById(R.id.status);
```

```
  Button send=(Button)findViewById(R.id.send);

  send.setOnClickListener(onSend);

  prefs=PreferenceManager.getDefaultSharedPreferences(this);
  prefs.registerOnSharedPreferenceChangeListener(prefListener);
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
  new MenuInflater(getApplication())
                                .inflate(R.menu.option, menu);

  return(super.onCreateOptionsMenu(menu));
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.prefs) {
    startActivity(new Intent(this, EditPreferences.class));

    return(true);
  }

  return(super.onOptionsItemSelected(item));
}

synchronized private Twitter getClient() {
  if (client==null) {
    client=new Twitter(prefs.getString("user", ""),
                       prefs.getString("password", ""));
  }

  return(client);
}

synchronized private void resetClient() {
  client=null;
}

private void updateStatus() {
  try {
    getClient().updateStatus(status.getText().toString());
  }
  catch (Throwable t) {
    Log.e("Patchy", "Exception in updateStatus()", t);
    goBlooey(t);
  }
}

private void goBlooey(Throwable t) {
  AlertDialog.Builder builder=new AlertDialog.Builder(this);
```

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

  private View.OnClickListener onSend=new View.OnClickListener() {
    public void onClick(View v) {
      updateStatus();
    }
  };

  private SharedPreferences.OnSharedPreferenceChangeListener prefListener=
    new SharedPreferences.OnSharedPreferenceChangeListener() {
    public void onSharedPreferenceChanged(SharedPreferences sharedPrefs, String
key) {
      if (key.equals("user") || key.equals("password")) {
        resetClient();
      }
    }
  };
}
```

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Add a checkbox to toggle whether the activity is creating a direct message or a status update. If the checkbox is checked, enable a `Spinner` containing the list of the user's followers (obtained from `getFollowers()`). For a direct message, use `sendMessage()` on the `Twitter` object. You may need to "follow" some classmates for everyone to have followers in their accounts to direct message.
- Use the `getStatus()` method on `Twitter` to display the user's current status at the top of the activity when the activity is opened. Re-fetch the status one second after submitting a status update, so the current status shows Twitter's rendition of your status – a good way to make sure the round-trip of status information is working as you expect.
- Store the password in an encrypted form in the user preferences, so that if somebody hacked a user's phone and got the preference store, they would still need the encryption key to find out the user's Twitter account password.

Listening To Your Friends

In this tutorial, we will start watching our friends' Twitter timelines, polling them for updates in a background thread managed by a service.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `18-JAR` edition of `Patchy` to use as a starting point.

Step #1: Create a Service Stub

Since we are creating a service to monitor information related to our Twitter account, let us create a service named `TwitterMonitor`. Create a file named `Patchy/src/apt/tutorial/two/TwitterMonitor.java` with the following content:

```
package apt.tutorial.two;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;

public class TwitterMonitor extends Service {
  @Override
  public void onCreate() {
    super.onCreate();
```

```
  }

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

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

We also need to add our service to `AndroidManifest.xml`, so add `<service android:name = ".TwitterMonitor" />` inside the `<application>` element.

At this point, your project will still compile and run, though our service is not yet doing anything.

Step #2: Set Up a Background Thread

Next, we need a well-managed background thread that can do our Twitter polling for us.

First, add an import for `java.util.concurrent.atomic.AtomicBoolean`, along with a private `AtomicBoolean` data member named `active`. Initialize this to `true`.

Also, define a static int value named `POLL_PERIOD` to be the amount of time you want between Twitter polling operations. Please make this no less than `60000` (one minute in milliseconds).

Then, create a `Runnable` named `threadBody` that will loop, sleeping `POLL_PERIOD` each pass, until `active` is toggled to `false`:

```
private Runnable threadBody=new Runnable() {
  public void run() {
    while (active.get()) {
      // do something with Twitter

      SystemClock.sleep(POLL_PERIOD);
```

```
    }
  }
};
```

In `onCreate()`, add a statement to start up our background thread on `threadBody`:

```
new Thread(threadBody).start();
```

Finally, in `onDestroy()`, set the `active` flag to `false`. Once the background thread wakes up from its current sleep cycle, it will see the `false` flag and fall out of its loop, terminating its thread:

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

  active.set(false);
}
```

Step #3: Poll Your Friends

Now, we have a service that will do work in the background. We just need to set up the work itself.

What we want to do is load our timeline (with our status and those from our friends) every poll period and find out those that are new. To do this, we need to create a `Twitter` object, use it to load the timeline, track the already-seen status messages, and identify the new ones. Right now, let us focus on loading the timeline.

First, add an import to `winterwell.jtwitter.Twitter` so we can create `Twitter` objects. We will also need `java.util.List` in a moment, so add an import for it as well.

Then, in our `threadBody` loop, add a call to a `poll()` method:

```
private Runnable threadBody=new Runnable() {
  public void run() {
```

```
    while (active.get()) {
      poll();
      SystemClock.sleep(POLL_PERIOD);
    }
  }
};
```

Finally, add this as the implementation of `poll()` in `TwitterMonitor`:

```
private void poll() {
  Twitter client=new Twitter();  // need credentials!
  List<Twitter.Status> timeline=client.getFriendsTimeline();
}
```

Note that this will not actually compile, since we do not have our Twitter user name or password. We will get that from the `Patchy` client of our service in the next tutorial. For the moment, ignore that compiler error, or hard-wire in account information until the next tutorial.

Right now, all we are doing in `poll()` is creating a fresh `Twitter` object and using it to load the timeline via `getFriendsTimeline()`.

Step #4: Find New Statuses

Once we have our timeline, we need to figure out which of the statuses are new. On the first poll, all will be new; subsequent polls should have a handful of new statuses, if any.

For the time being, let us track the already-seen status IDs via a `Set<Long>`, so add the following data member (and the `HashSet` and `Set` imports to match):

```
private Set<Long> seenStatus=new HashSet<Long>();
```

Then, we can walk the timeline, see if each status ID is in the `Set`, and process those that are not, via this change to `poll()`:

```
private void poll() {
  Twitter client=new Twitter();  // need credentials!
  List<Twitter.Status> timeline=client.getFriendsTimeline();
```

```
  for (Twitter.Status s : timeline) {
    if (!seenStatus.contains(s.id)) {
      // found a new one!
      seenStatus.add(s.id);
    }
  }
}
```

Of course, eventually, we will need to alert `Patchy` to the new statuses, but we can leave that for the next tutorial.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Add a preference for the timeline polling period. Use that polling period in the service, including changing the polling period when the preference changes.
- If you added direct-message support in the previous tutorial, have the service also poll for changes to the list of followers, to eventually control the contents of your direct-message `Spinner`.

No, Really Listening To Your Friends

In the previous tutorial, we set up a service to watch for new timeline entries from our friends. The service merely finds out about these new statuses – now we need to get those someplace for our activity to display them.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `19-CreateService` edition of `Patchy` to use as a starting point.

Step #1: Make the Service a Singleton

When running a local service, we do not need to mess around with IPC or AIDL, since everything is in the same process. However, we do have to have a way to communicate with that service. The easiest solution is to make it a singleton, accessible from a static method on the service class.

With that in mind, create a static data member on `TwitterMonitor` for the singleton:

```
public static TwitterMonitor singleton=null;
```

Initialize the singleton in `onCreate()` and reset it to null in `onDestroy()`. Forget the latter, and you may wind up with a memory leak!

Step #2: Defining the Callback

We need to provide some sort of callback to the service, so our activity can be notified when new status updates are available. With that in mind, create an `ITwitterListener` interface as follows:

```
package apt.tutorial.two;

public interface ITwitterListener {
  void newFriendStatus(String friend, String status,
                       String createdAt);
}
```

Step #3: Enable Callbacks in the Service

Now we need to set up a means for the service client (`Patchy`) to start the service and register an `ITwitterListener` that will be invoked when a new status arrives. The callback will be associated with a user name and password, so we have the credentials with which to call Twitter via its API.

First, let us create an inner class in `TwitterMonitor` to represent the callback and login credentials. Add this implementation of `Account` to your `TwitterMonitor`:

```
class Account {
  String user=null;
  String password=null;
  ITwitterListener callback=null;

  Account(String user, String password,
          ITwitterListener callback) {
    this.user=user;
    this.password=password;
    this.callback=callback;
  }
}
```

Next, import `java.util.Map` and `java.util.concurrent.ConcurrentHashMap` and set up a private data member representing the roster of outstanding accounts:

```
private Map<ITwitterListener, Account> accounts=
        new ConcurrentHashMap<ITwitterListener, Account>();
```

Now, we can set up public APIs on our service for our client to use to register and remove an account:

```
public void registerAccount(String user, String password,
                            ITwitterListener callback) {
  Account l=new Account(user, password, callback);

  poll(l);
  accounts.put(callback, l);
}

public void removeAccount(ITwitterListener callback) {
  accounts.remove(callback);
}
```

Notice that we call `poll()` with our `Account` when it is registered. This will fetch our current timeline right away, rather than wait for our polling loop to cycle back around. The downside is that this `poll()` is called on the main thread rather than a background thread – we will address this in a later tutorial.

Next, update our `threadBody` to make use of our roster of `Account` objects:

```
private Runnable threadBody=new Runnable() {
  public void run() {
    while (active.get()) {
      for (Account l : accounts.values()) {
        poll(l);
      }

      SystemClock.sleep(POLL_PERIOD);
    }
  }
};
```

This too calls `poll()` with the `Account` information. Hence, we need to update `poll()` to use the `Account` and call the callback on new status messages:

```
private void poll(Account l) {
  try {
    Twitter client=new Twitter(l.user, l.password);
    List<Twitter.Status> timeline=client.getFriendsTimeline();

    for (Twitter.Status s : timeline) {
      if (!seenStatus.contains(s.id)) {
        l.callback.newFriendStatus(s.user.screenName, s.text,
                                   s.createdAt.toString());
        seenStatus.add(s.id);
      }
    }
  }
  catch (Throwable t) {
    android.util.Log.e("TwitterMonitor",
                       "Exception in poll()", t);
  }
}
```

Step #4: Manage the Service and Register the Account

Next, we need to take steps in `Patchy` to connect to the `TwitterMonitor` service and arrange to get callbacks when new timeline entries are ready.

First, modify `onCreate()` in `Patchy` to start the service and, after a two-second delay to let the service start up, registers our account information and an `ITwitterListener` instance:

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

  status=(EditText)findViewById(R.id.status);

  Button send=(Button)findViewById(R.id.send);

  send.setOnClickListener(onSend);

  prefs=PreferenceManager.getDefaultSharedPreferences(this);
  prefs.registerOnSharedPreferenceChangeListener(prefListener);
```

```
  startService(new Intent(this, TwitterMonitor.class));

  send.postDelayed(new Runnable() {
    public void run() {
      if (TwitterMonitor.singleton!=null) {
        TwitterMonitor.singleton.registerAccount(prefs.getString("user", ""),
                    prefs.getString("password", ""),
                    listener);
      }
    }
  }, 2000);
}
```

The two-second delay is a hack, because we do not know if the service is ready in 50 milliseconds or not yet ready even after two seconds. In the next tutorial, when we move `TwitterMonitor` into a remote service, we will resolve this problem.

We need to stop our service, after removing our account, when we are done, so add an implementation of `onDestroy()` to `Patchy` as follows:

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

  TwitterMonitor.singleton.removeAccount(listener);
  stopService(new Intent(this, TwitterMonitor.class));
}
```

Our code in `onCreate()` refers to a listener instance, so add a definition of it as an `ITwitterListener` implementation to `Patchy`:

```
private ITwitterListener listener=new ITwitterListener() {
  public void newFriendStatus(String friend, String status,
                              String createdAt) {
  }
};
```

Right now, this listener does not do anything – we will address that minor shortcoming in the next section.

At this point, we register our account at startup and remove it on shutdown. What we are missing is account changes. If the user, via the preferences,

adjusts the user name or password, we need to get that information to the service. The cleanest way to do that is to remove our current account and register a fresh one.

With that in mind, modify `resetClient()` to handle that chore:

```
synchronized private void resetClient() {
  client=null;
  TwitterMonitor.singleton.removeAccount(listener);
  TwitterMonitor.singleton.registerAccount(prefs.getString("user", ""),
                      prefs.getString("password", ""),
                      listener);
}
```

Step #5: Display the Timeline

At this point, `Patchy` is fully integrated with the service and should receive callbacks on timeline changes. However, it does not do anything with those changes. So, we need to add something to our UI to actually display the timeline. One easy way to do that is via a `ListView` and custom row layout – that way, the timeline can be as long as we want, and it will scroll to show all of the entries.

So, modify `Patchy/res/layout/main.xml` to add a `ListView`, as shown below:

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:stretchColumns="1"
  >
  <TableRow>
    <TextView android:text="Status:" />
    <EditText android:id="@+id/status"
      android:singleLine="false"
      android:gravity="top"
      android:lines="5"
      android:scrollHorizontally="false"
      android:maxLines="5"
      android:maxWidth="200sp"
    />
  </TableRow>
  <Button android:id="@+id/send"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Send"
```

```
  />
  <ListView android:id="@+id/timeline"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
  />
</TableLayout>
```

We also need to create a layout for our rows. We have three pieces of information to display: the screen name of the friend, the status message, and the date the status was modified. Create `Patchy/res/layout/row.xml` with the following layout to display all three of those pieces:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:orientation="vertical"
  android:padding="4px"
  >
  <LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:layout_weight="1"
    android:gravity="center_vertical"
    >
    <TextView android:id="@+id/friend"
      android:layout_width="0px"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:gravity="left"
      android:textStyle="bold"
      android:singleLine="true"
      android:ellipsize="end"
    />
    <TextView android:id="@+id/created_at"
      android:layout_width="0px"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:gravity="right"
      android:singleLine="true"
      android:ellipsize="end"
    />
  </LinearLayout>
  <TextView android:id="@+id/status"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:gravity="center_vertical"
    android:singleLine="false"
  />
</LinearLayout>
```

Now we need to replicate a lot of the logic from `LunchList` to populate this custom list in `Patchy`.

Add a `TimelineEntry` inner class to `Patchy` to hold a single timeline entry:

```
class TimelineEntry {
  String friend="";
  String createdAt="";
  String status="";

  TimelineEntry(String friend, String createdAt,
                String status) {
    this.friend=friend;
    this.createdAt=createdAt;
    this.status=status;
  }
}
```

Now, add an `ArrayList` of `TimelineEntry` objects to serve as the timeline itself, as a private data member of Patchy:

```
private List<TimelineEntry> timeline=new ArrayList<TimelineEntry>();
```

Then, add a `TimelineEntryWrapper` that will update one of our rows with the contents of a `TimelineEntry`:

```
class TimelineEntryWrapper {
  private TextView friend=null;
  private TextView createdAt=null;
  private TextView status=null;
  private View row=null;

  TimelineEntryWrapper(View row) {
    this.row=row;
  }

  void populateFrom(TimelineEntry s) {
    getFriend().setText(s.friend);
    getCreatedAt().setText(s.createdAt);
    getStatus().setText(s.status);
  }

  TextView getFriend() {
    if (friend==null) {
      friend=(TextView)row.findViewById(R.id.friend);
    }

    return(friend);
```

```
  }

  TextView getCreatedAt() {
    if (createdAt==null) {
      createdAt=(TextView)row.findViewById(R.id.created_at);
    }

    return(createdAt);
  }

  TextView getStatus() {
    if (status==null) {
      status=(TextView)row.findViewById(R.id.status);
    }

    return(status);
  }
}
```

Next, add a `TimelineAdapter` implementation that will display our timeline:

```
class TimelineAdapter extends ArrayAdapter<TimelineEntry> {
  TimelineAdapter() {
    super(Patchy.this, R.layout.row, timeline);
  }

  public View getView(int position, View convertView,
                      ViewGroup parent) {
    View row=convertView;
    TimelineEntryWrapper wrapper=null;

    if (row==null) {
      LayoutInflater inflater=getLayoutInflater();

      row=inflater.inflate(R.layout.row, null);
      wrapper=new TimelineEntryWrapper(row);
      row.setTag(wrapper);
    }
    else {
      wrapper=(TimelineEntryWrapper)row.getTag();
    }

    wrapper.populateFrom(timeline.get(position));

    return(row);
  }
}
```

We need to have a `TimelineAdapter` as a private data member, so add one (initialized to null), then instantiate the adapter in `onCreate()`:

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

  status=(EditText)findViewById(R.id.status);

  Button send=(Button)findViewById(R.id.send);

  send.setOnClickListener(onSend);

  prefs=PreferenceManager.getDefaultSharedPreferences(this);
  prefs.registerOnSharedPreferenceChangeListener(prefListener);

  startService(new Intent(this, TwitterMonitor.class));

  adapter=new TimelineAdapter();
  ((ListView)findViewById(R.id.timeline)).setAdapter(adapter);

  send.postDelayed(new Runnable() {
    public void run() {
      if (TwitterMonitor.singleton!=null) {
        TwitterMonitor.singleton.registerAccount(prefs.getString("user", ""),
                    prefs.getString("password", ""),
                    listener);
      }
    }
  }, 2000);
}
```

Finally, add logic to our listener object to create a `TimelineEntry` and add it to the top of the `TimelineAdapter`, so new entries are added to the head of the list:

```
private ITwitterListener listener=new ITwitterListener() {
  public void newFriendStatus(String friend, String status,
                              String createdAt) {
    runOnUiThread(new Runnable() {
      public void run() {
        adapter.insert(new TimelineEntry(friend,
                                         createdAt,
                                         status),
                       0);
      }
    })
  }
};
```

With all that behind you, recompile and reinstall the application. Now, `Patchy` will display your timeline (after a two-second delay) and should keep the timeline current as new entries roll in:

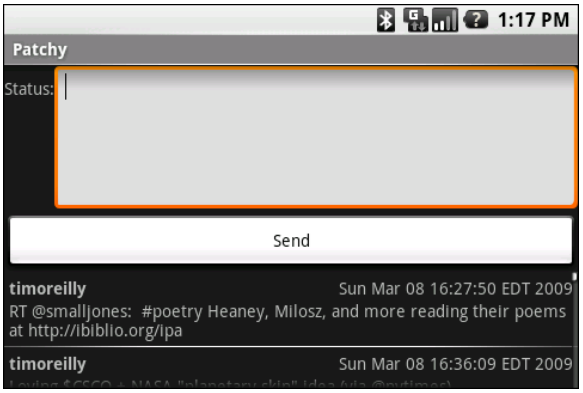


Figure 37. The timeline as displayed in Patchy

Here is a full implementation of `Patchy` after all of the above changes:

```
package apt.tutorial.two;

import android.app.Activity;
import android.app.AlertDialog;
import android.content.Intent;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.preference.PreferenceManager;
import android.view.LayoutInflater;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.view.ViewGroup;
import android.util.Log;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ListView;
import android.widget.TextView;
import java.util.ArrayList;
import java.util.List;
import winterwell.jtwitter.Twitter;

public class Patchy extends Activity {
  private EditText status=null;
  private SharedPreferences prefs=null;
  private Twitter client=null;
```

```
  private List<TimelineEntry> timeline=new ArrayList<TimelineEntry>();
  private TimelineAdapter adapter=null;

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

    status=(EditText)findViewById(R.id.status);

    Button send=(Button)findViewById(R.id.send);

    send.setOnClickListener(onSend);

    prefs=PreferenceManager.getDefaultSharedPreferences(this);
    prefs.registerOnSharedPreferenceChangeListener(prefListener);

    startService(new Intent(this, TwitterMonitor.class));

    adapter=new TimelineAdapter();
    ((ListView)findViewById(R.id.timeline)).setAdapter(adapter);

    send.postDelayed(new Runnable() {
      public void run() {
        if (TwitterMonitor.singleton!=null) {
          TwitterMonitor.singleton.registerAccount(prefs.getString("user", ""),
                    prefs.getString("password", ""),
                    listener);
        }
      }
    }, 2000);
  }

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

    TwitterMonitor.singleton.removeAccount(listener);
    stopService(new Intent(this, TwitterMonitor.class));
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    new MenuInflater(getApplication())
                                  .inflate(R.menu.option, menu);

    return(super.onCreateOptionsMenu(menu));
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.prefs) {
      startActivity(new Intent(this, EditPreferences.class));
```

```
      return(true);
    }

    return(super.onOptionsItemSelected(item));
  }

  synchronized private Twitter getClient() {
    if (client==null) {
      client=new Twitter(prefs.getString("user", ""),
                         prefs.getString("password", ""));
    }

    return(client);
  }

  synchronized private void resetClient() {
    client=null;
    TwitterMonitor.singleton.removeAccount(listener);
    TwitterMonitor.singleton.registerAccount(prefs.getString("user", ""),
                         prefs.getString("password", ""),
                         listener);
  }

  private void updateStatus() {
    try {
      getClient().updateStatus(status.getText().toString());
    }
    catch (Throwable t) {
      Log.e("Patchy", "Exception in updateStatus()", t);
      goBlooey(t);
    }
  }

  private void goBlooey(Throwable t) {
    AlertDialog.Builder builder=new AlertDialog.Builder(this);

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

  private View.OnClickListener onSend=new View.OnClickListener() {
    public void onClick(View v) {
      updateStatus();
    }
  };

  private SharedPreferences.OnSharedPreferenceChangeListener prefListener=
    new SharedPreferences.OnSharedPreferenceChangeListener() {
    public void onSharedPreferenceChanged(SharedPreferences sharedPrefs,
                                          String key) {
      if (key.equals("user") || key.equals("password")) {
```

```
      resetClient();
    }
  }
};

private ITwitterListener listener=new ITwitterListener() {
  public void newFriendStatus(String friend, String status,
                              String createdAt) {
    runOnUiThread(new Runnable() {
      public void run() {
        adapter.insert(new TimelineEntry(friend,
                                         createdAt,
                                         status),
                       0);
      }
    })
  }
};

class TimelineEntry {
  String friend="";
  String createdAt="";
  String status="";

  TimelineEntry(String friend, String createdAt,
                String status) {
    this.friend=friend;
    this.createdAt=createdAt;
    this.status=status;
  }
}

class TimelineAdapter extends ArrayAdapter<TimelineEntry> {
  TimelineAdapter() {
    super(Patchy.this, R.layout.row, timeline);
  }

  public View getView(int position, View convertView,
                      ViewGroup parent) {
    View row=convertView;
    TimelineEntryWrapper wrapper=null;

    if (row==null) {
      LayoutInflater inflater=getLayoutInflater();

      row=inflater.inflate(R.layout.row, null);
      wrapper=new TimelineEntryWrapper(row);
      row.setTag(wrapper);
    }
    else {
      wrapper=(TimelineEntryWrapper)row.getTag();
    }

    wrapper.populateFrom(timeline.get(position));
```

```
      return(row);
    }
  }

  class TimelineEntryWrapper {
    private TextView friend=null;
    private TextView createdAt=null;
    private TextView status=null;
    private View row=null;

    TimelineEntryWrapper(View row) {
      this.row=row;
    }

    void populateFrom(TimelineEntry s) {
      getFriend().setText(s.friend);
      getCreatedAt().setText(s.createdAt);
      getStatus().setText(s.status);
    }

    TextView getFriend() {
      if (friend==null) {
        friend=(TextView)row.findViewById(R.id.friend);
      }

      return(friend);
    }

    TextView getCreatedAt() {
      if (createdAt==null) {
        createdAt=(TextView)row.findViewById(R.id.created_at);
      }

      return(createdAt);
    }

    TextView getStatus() {
      if (status==null) {
        status=(TextView)row.findViewById(R.id.status);
      }

      return(status);
    }
  }
}
```

Similarly, here is a full implementation of `TwitterMonitor` as of this point:

```
package apt.tutorial.two;

import android.app.Service;
import android.content.Intent;
```

```
import android.os.IBinder;
import android.os.SystemClock;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicBoolean;
import winterwell.jtwitter.Twitter;

public class TwitterMonitor extends Service {
  public static TwitterMonitor singleton=null;
  private static final int POLL_PERIOD=60000;
  private AtomicBoolean active=new AtomicBoolean(true);
  private Set<Long> seenStatus=new HashSet<Long>();
  private Map<ITwitterListener, Account> accounts=
          new ConcurrentHashMap<ITwitterListener, Account>();

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

    singleton=this;
    new Thread(threadBody).start();
  }

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

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

    active.set(false);
    singleton=null;
  }

  public void registerAccount(String user, String password,
                              ITwitterListener callback) {
    Account l=new Account(user, password, callback);

    poll(l);
    accounts.put(callback, l);
  }

  public void removeAccount(ITwitterListener callback) {
    accounts.remove(callback);
  }

  private void poll(Account l) {
    try {
      Twitter client=new Twitter(l.user, l.password);
```

```
      List<Twitter.Status> timeline=client.getFriendsTimeline();

      for (Twitter.Status s : timeline) {
        if (!seenStatus.contains(s.id)) {
          l.callback.newFriendStatus(s.user.screenName, s.text,
                                     s.createdAt.toString());
          seenStatus.add(s.id);
        }
      }
    }
    catch (Throwable t) {
      android.util.Log.e("TwitterMonitor",
                         "Exception in poll()", t);
    }
  }

  private Runnable threadBody=new Runnable() {
    public void run() {
      while (active.get()) {
        for (Account l : accounts.values()) {
          poll(l);
        }

        SystemClock.sleep(POLL_PERIOD);
      }
    }
  };

  class Account {
    String user=null;
    String password=null;
    ITwitterListener callback=null;

    Account(String user, String password,
            ITwitterListener callback) {
      this.user=user;
      this.password=password;
      this.callback=callback;
    }
  }
}
```

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Switch to using a database as the storage mechanism for statuses, rather than RAM. The service can insert new statuses into the database and keep the table trimmed to some maximum number of statuses (to keep storage requirements down). The callback would

notify the activity that there were updates requiring a refresh of the activity's `CursorAdapter` on the database. Consider using a preference to allow the user to control the maximum number of statuses to track.

- Right now, when several timeline entries are added through one Twitter API call (e.g., the initial poll), they are put in the list in the wrong order. Change `Patch` and `TwitterMonitor` to have them appear in the proper order, no matter how many entries are retrieved in a single poll.
- If you added direct-message support in the previous tutorials, have the service also invoke the callback on a change to the follower roster, and have that affect the follower `Spinner` contents.
- Have the background thread running only if there is one or more registered listeners.

Your Friends Seem Remote

In the previous example, we implemented the Twitter friends timeline via a local service. Now, let us reimplement this as a remote service, so we can allow other applications to possibly use our Twitter component.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `20-LocalService` edition of `Patchy` to use as a starting point.

Step #1: Create a Fresh Project

First, we should create a separate project for the `TwitterMonitor`, to simulate it being some third-party component and to force it to run in a separate process as a separate user.

So, via `activitycreator` or Eclipse, create a `TMonitor` project peer of `Patchy`, using the package `apt.tutorial.three` (so `TMonitor` and `Patchy` can both be on the emulator at the same time).

Step #2: Move the Service to the New Project

Now, let us move the `TwitterMonitor` code from `Patchy` into the new `TMonitor` project.

First, move `Patchy/src/apt/tutorial/two/TwitterMonitor.java` to `TMonitor/src/apt/tutorial/three`, replacing the automatically-generated `TwitterMonitor.java`. Also, go in and fix up the package statement, reflecting that the new `TwitterMonitor` is in apt.tutorial.three.

Then, since both `Patchy` and `TwitterMonitor` need the JTwitter JAR, copy that jar from `Patchy/libs/` to `TMonitor/libs/`.

Next, modify `TMonitor/AndroidManifest.xml` to get rid of the `<activity>` element (since `TMonitor` has no activity) and replace it with the `<service>` element from `Patchy/AndroidManifest.xml`. While you have `Patchy/AndroidManifest.xml` open, get rid of that `<service>` element after you have put it in the `TMonitor/AndroidManifest.xml` file.

Finally, since we removed the `TwitterMonitor` class from `Patchy`, we need to get rid of the compiled edition of that class. If you are using Eclipse, force a rebuild of your project; otherwise, get rid of `Patchy/bin/`, so on the next recompile, it will not have any of the old code lying around.

At this point, if you recompile each project, you will see they both have compiler errors – `TwitterMonitor` cannot find `ITwitterListener` and `Patchy` cannot find `TwitterMonitor`.

Step #3: Implement and Copy the AIDL

Now, we need to implement AIDL to represent the interface that the service exposes to the client and the callback that the client exposes to the service.

We already almost have the callback AIDL, since AIDL closely resembles Java interfaces. Move `Patchy/src/apt/tutorial/two/ITwitterListener.java` to

Patchy/src/apt/tutorial/ITwitterListener.aidl, moving it up one directory and changing the file extension to .aidl. Then, remove the public keywords, and the AIDL is set.

Then, copy the new ITwitterListener.aidl file to the corresponding directory in TMonitor (TMonitor/src/apt/tutorial/ITwitterListener.aidl). In TwitterMonitor, add the import to apt.tutorial.ITwitterListener. We also have to deal with the possibility of an android.os.RemoteException when using the callback, since the calling process might have unexpectedly terminated. So, we need to wrap our callback use in a try/catch block as follows:

```
private void poll(Account l) {
  try {
    Twitter client=new Twitter(l.user, l.password);
    List<Twitter.Status> timeline=client.getFriendsTimeline();

    for (Twitter.Status s : timeline) {
      if (!seenStatus.contains(s.id)) {
        try {
          l.callback.newFriendStatus(s.user.screenName, s.text,
                                     s.createdAt.toString());
          seenStatus.add(s.id);
        }
        catch (Throwable t) {
          Log.e("TwitterMonitor", "Exception in callback", t);
        }
      }
    }
```

At this point, the TMonitor project should compile cleanly. However, we still need to develop the AIDL for exposing the service APIs to the client.

To that end, create TMonitor/src/apt/tutorial/ITwitterMonitor.aidl with the following content:

```
package apt.tutorial;

import apt.tutorial.ITwitterListener;

interface ITwitterMonitor {
  void registerAccount(String user, String password,
                       ITwitterListener callback);
  void removeAccount(ITwitterListener callback);
}
```

Step #4: Implement the Client Side

Next, we need to fix up the `Patchy` client, so it uses the new `ITwitterMonitor` service interface instead of attempting to access a local `TwitterMonitor` singleton.

First, add an import for `apt.tutorial.ITwitterListener`, since we moved that file out of the apt.tutorial.two package. You will probably need to clean your project again (e.g., `rm -rf bin/`) to get rid of the old compiled classes.

Then, we need to make one slight modification to our `ITwitterListener` implementation: our anonymous inner class actually has to implement `ITwitterListener.Stub` instead of `ITwitterListener`. We should also change it to use `runOnUiThread()`, since we do not know what thread our callback will be invoked upon. So, change the `listener` implementation to:

```
private ITwitterListener listener=new ITwitterListener.Stub() {
  public void newFriendStatus(final String friend,
                              final String status,
                              final String createdAt) {
    runOnUiThread(new Runnable() {
      public void run() {
        adapter.insert(new TimelineEntry(friend,
                                         createdAt,
                                         status), 0);
      }
    });
  }
};
```

Next, copy `TMonitor/src/apt/tutorial/ITwitterMonitor.aidl` to `Patchy/src/apt/tutorial`, so the AIDL is available in both projects. Also, add an import of `apt.tutorial.ITwitterMonitor` to `Patchy` and an `ITwitterMonitor` data member named service.

Next, add a `ServiceConnection` data member named `svcConn` with the following implementation, which moves the initial `registerCallback()` API call from `onCreate()` to here:

```
private ServiceConnection svcConn=new ServiceConnection() {
  public void onServiceConnected(ComponentName className,
```

```
                                IBinder binder) {
    service=ITwitterMonitor.Stub.asInterface(binder);

    try {
      service.registerAccount(prefs.getString("user", ""),
                              prefs.getString("password", ""),
                              listener);
    }
    catch (Throwable t) {
      Log.e("Patchy", "Exception in call to registerAccount()", t);
      goBlooey(t);
    }
  }

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

Hence, now we register our callback as soon as our service connection is ready.

Everywhere in the code, change all `TwitterMonitor.singleton` references to use `service` instead.

In `onCreate()`, get rid of the `startService()` and `registerCallback()` calls, replacing them with a `bindService()` call:

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

  status=(EditText)findViewById(R.id.status);

  Button send=(Button)findViewById(R.id.send);

  send.setOnClickListener(onSend);

  prefs=PreferenceManager.getDefaultSharedPreferences(this);
  prefs.registerOnSharedPreferenceChangeListener(prefListener);

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

  adapter=new TimelineAdapter();
  ((ListView)findViewById(R.id.timeline)).setAdapter(adapter);
}
```

Similarly, in `onDestroy()`, get rid of the `stopService()` call, replacing it with `unbindService()`. Also, wrap the `removeCallback()` in a `try/catch` block, to deal with the possibility that our connection to the `TwitterMonitor` service has been broken for some reason:

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

  try {
    service.removeAccount(listener);
  }
  catch (Throwable t) {
    Log.e("Patchy", "Exception in call to removeAccount()", t);
    goBlooey(t);
  }

  unbindService(svcConn);
}
```

We also need to change `resetClient()` to wrap its service API calls in a `try/catch` block, as follows:

```
synchronized private void resetClient() {
  client=null;

  try {
    service.removeAccount(listener);
    service.registerAccount(prefs.getString("user", ""),
                            prefs.getString("password", ""),
                            listener);
  }
  catch (Throwable t) {
    Log.e("Patchy", "Exception in resetClient()", t);
    goBlooey(t);
  }
}
```

At this point, the `Patchy` project should compile cleanly. It will not run, though, without the corresponding service.

Step #5: Implement the Service Side

Finally, we need to make similar sorts of changes on the `TwitterMonitor` service, to implement the new `ITwitterMonitor` interface and make it available to `Patchy`.

First, modify `TMonitor/AndroidManifest.xml` to add an `<intent-filter>` (so `Patchy` can reference it from another process and package) and add the `INTERNET` permission that JTwitter will need:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="apt.tutorial.three"
      android:versionCode="1"
      android:versionName="1.0">
  <uses-permission android:name="android.permission.INTERNET" />
  <application android:label="@string/app_name">
    <service android:name=".TwitterMonitor">
      <intent-filter>
        <action android:name="apt.tutorial.ITwitterMonitor" />
      </intent-filter>
    </service>
  </application>
</manifest>
```

Next, rename the `TwitterMonitor` methods `registerAccount()` and `removeAccount()` to `registerAccountImpl()` and `removeAccountImpl()`.

Then, add an `ITwitterMonitor.Stub` anonymous inner class instance as follows:

```
private final ITwitterMonitor.Stub binder=new ITwitterMonitor.Stub() {
  public void registerAccount(String user, String password,
                              ITwitterListener callback) {
    registerAccountImpl(user, password, callback);
  }

  public void removeAccount(ITwitterListener callback) {
    removeAccountImpl(callback);
  }
};
```

All we do is delegate the public API to our `...Impl()` implementations.

We also need to return this `ITwitterMonitor.Stub` instance from `onBind()`:

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

Finally, we need to do something about our polling logic, so when we register an account we do not tie up the UI thread of the client doing an immediate `poll()`, yet we still quickly get Twitter status updates rather than waiting a full minute. To help resolve this, we will use two polling periods: a one-second period when we have no accounts, and the original one-minute period for when we do have accounts. That way, we will find out about our first account within one second of it being registered; only then will we slow down, so as not to abuse the Twitter service.

To that end, add the following data members to `TwitterMonitor`:

```
private static final int INITIAL_POLL_PERIOD=1000;
private int pollPeriod=INITIAL_POLL_PERIOD;
```

Also, change `threadBody` to use the new polling logic:

```
private Runnable threadBody=new Runnable() {
  public void run() {
    while (active.get()) {
      for (Account l : accounts.values()) {
        poll(l);
        pollPeriod=POLL_PERIOD;
      }

      SystemClock.sleep(pollPeriod);
    }
  }
};
```

Now, you should be able to recompile and reinstall both projects. Launching `Patchy` will give you similar results as before – the difference being that now it is using the remote `TwitterMonitor` service, rather than a local one.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Design two other applications that could use Twitter services (perhaps beyond those you have already implemented) and could take advantage of a common Twitter component.
- If you added direct-message support in the previous tutorials, convert the follower roster callback to use AIDL as well.

BFF

Right now, the only way you find out about status updates from your Twitter friends is by looking at your application. However, perhaps there are certain friends who you want a more proactive way to find out they have updated their status. In this tutorial, we will set up our remote service to support this "BFF" concept and raise a `Notification` when these friends change their status.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `21-RemoteService` edition of `Patchy` to use as a starting point.

Step #1: Create the BFF Tab

First, let us convert `Patchy` into a two-tab UI, one tab with our current contents, one tab with a list of our friends (i.e., who we are following on Twitter).

Change the `Patchy/res/layout/main.xml` layout to look like this:

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

```
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <TabHost android:id="@android:id/tabhost"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <TabWidget android:id="@android:id/tabs"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
    />
    <FrameLayout android:id="@android:id/tabcontent"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:paddingTop="62px"
      >
      <TableLayout android:id="@+id/status_tab"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:stretchColumns="1"
        >
        <TableRow>
          <TextView android:text="Status:" />
          <EditText android:id="@+id/status"
            android:singleLine="false"
            android:gravity="top"
            android:lines="5"
            android:scrollHorizontally="false"
            android:maxLines="5"
            android:maxWidth="200sp"
          />
        </TableRow>
        <Button android:id="@+id/send"
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="Send"
        />
        <ListView android:id="@+id/timeline"
          android:layout_width="fill_parent"
          android:layout_height="fill_parent"
        />
      </TableLayout>
      <ListView android:id="@+id/friends"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
      />
    </FrameLayout>
  </TabHost>
</LinearLayout>
```

Then, have `Patchy` extend from `TabActivity` instead of `Activity`.

Finally, we will cheat and modify `onCreate()` to both add the two tabs and to make the call via JTwitter to get the list of friends with which to populate the list. This is cheating insofar as it makes the application slow to launch – a better implementation would put this in the background thread. However, to keep things simple, modify `onCreate()` to look like this:

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

  status=(EditText)findViewById(R.id.status);

  Button send=(Button)findViewById(R.id.send);

  send.setOnClickListener(onSend);

  prefs=PreferenceManager.getDefaultSharedPreferences(this);
  prefs.registerOnSharedPreferenceChangeListener(prefListener);

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

  adapter=new TimelineAdapter();
  ((ListView)findViewById(R.id.timeline)).setAdapter(adapter);

  TabHost.TabSpec spec=getTabHost().newTabSpec("tag1");

  spec.setContent(R.id.status_tab);
  spec.setIndicator("Status", getResources()
                              .getDrawable(R.drawable.status));
  getTabHost().addTab(spec);

  spec=getTabHost().newTabSpec("tag2");
  spec.setContent(R.id.friends);
  spec.setIndicator("Friends", getResources()
                               .getDrawable(R.drawable.friends));
  getTabHost().addTab(spec);

  getTabHost().setCurrentTab(0);

  for (Twitter.User u : getClient().getFriends()) {
    friends.add(u.screenName);
  }

  Collections.sort(friends);

  friendsList=(ListView)findViewById(R.id.friends);

  friendsAdapter=new ArrayAdapter<String>(this,
                    android.R.layout.simple_list_item_multiple_choice,
                    friends);
```

```
  friendsList.setAdapter(friendsAdapter);
  friendsList.setItemsCanFocus(false);
  friendsList.setChoiceMode(ListView.CHOICE_MODE_MULTIPLE);
}
```

Note that we are defining this list as being checkable, so users can choose one or several friends to be given BFF status.

Also note that you will also need to add some imports, plus some matching data members:

```
private List<String> friends=new ArrayList<String>();
private ArrayAdapter<String> friendsAdapter=null;
private ListView friendsList=null;
```

At this point, recompile and rebuild `Patchy` – when you run it, after a short delay, you should see two tabs, including one with a list of all our friends:

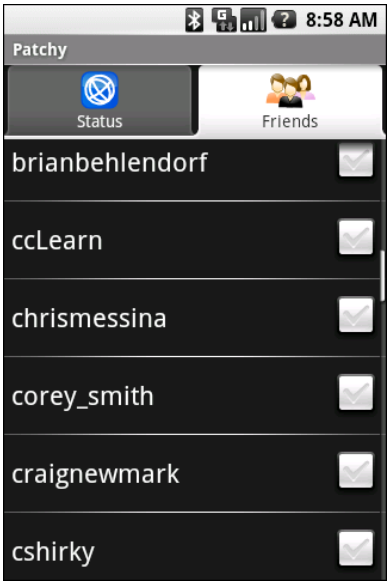


Figure 38. Your Twitter friends, as seen in Patchy

Step #2: Extend the AIDL for BFF

Our AIDL does not have a spot for the best-friends-forever roster, so modify `TMonitor/src/apt/tutorial/ITwitterMonitor.aidl` to include a new `setBestFriends()` method:

```
package apt.tutorial;

import apt.tutorial.ITwitterListener;

interface ITwitterMonitor {
  void registerAccount(String user, String password,
                       ITwitterListener callback);
  void removeAccount(ITwitterListener callback);
  void setBestFriends(ITwitterListener callback,
                      in List<String> bff);
}
```

For the moment, add an implementation of `setBestFriends()` in our `ITwitterMonitor.Stub` instance that does not do anything – we will add the actual logic later in this tutorial.

At this point, `TMonitor` should compile properly.

Also, copy `TMonitor/src/apt/tutorial/ITwitterMonitor.aidl` over to `Patchy/src/apt/tutorial/`, so both applications have the same AIDL definition.

Step #3: Update the Service on BFF Changes

We need to get the BFF list from `Patchy` to `TwitterMonitor`. The simplest approach is to add an option menu item for that, in `Patchy/res/menu/option.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/prefs"
    android:title="Settings"
    android:icon="@android:drawable/ic_menu_preferences"
  />
  <item android:id="@+id/bff"
    android:title="Set Best Friends"
```

```
    android:icon="@drawable/friends"
  />
</menu>
```

Then, we can implement the logic to call the service via our new API – replace the existing `onOptionsItemSelected()` implementation in `Patchy` to be:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.prefs) {
    startActivity(new Intent(this, EditPreferences.class));

    return(true);
  }
  else if (item.getItemId()==R.id.bff) {
    try {
      List<String> bff=new ArrayList<String>();

      for (int i=0;i<friends.size();i++) {
        if (friendsList.isItemChecked(i)) {
          bff.add(friends.get(i));
        }
      }

      service.setBestFriends(listener, bff);
    }
    catch (Throwable t) {
      Log.e("Patchy",
            "Exception in onOptionsItemSelected()", t);
      goBlooey(t);
    }

    return(true);
  }
```

Now, when we click the Set Best Friends menu option, the service will be told the new roster of best friends.

Step #4: Service Watches for BFF Status Changes

Now, we need to flesh out our implementation of `setBestFriends()` on the service, plus use it to find out when a BFF updates their Twitter status.

Add an import for `java.util.concurrent.CopyOnWriteArrayList` plus a `CopyOnWriteArrayList` data member named `bff` to `TwitterMonitor`. By using

`CopyOnWriteArrayList`, we can make changes to the list without disturbing any use of the list by the primary background thread.

Then, update our `ITwitterMonitor.Stub` implementation to look like this:

```
private final ITwitterMonitor.Stub binder=new ITwitterMonitor.Stub() {
  public void registerAccount(String user, String password,
                              ITwitterListener callback) {
    registerAccountImpl(user, password, callback);
  }

  public void removeAccount(ITwitterListener callback) {
    removeAccountImpl(callback);
  }

  public void setBestFriends(ITwitterListener callback,
                             List<String> newBff) {
    bff.clear();
    bff.addAll(newBff);
  }
};
```

Then, alter `poll()` to scan the list for hits on every new status update:

```
private void poll(Account l) {
  try {
    Twitter client=new Twitter(l.user, l.password);
    List<Twitter.Status> timeline=client.getFriendsTimeline();

    for (Twitter.Status s : timeline) {
      if (!seenStatus.contains(s.id)) {
        try {
          l.callback.newFriendStatus(s.user.screenName, s.text,
                                     s.createdAt.toString());
          seenStatus.add(s.id);
        }
        catch (Throwable t) {
          Log.e("TwitterMonitor", "Exception in callback", t);
        }

        if (bff.contains(s.user.screenName)) {
          notify(s.user.screenName);
        }
      }
    }
  }
  catch (Throwable t) {
    Log.e("TwitterMonitor", "Exception in poll()", t);
  }
}
```

Step #5: Raise a Notification

Notice how the `poll()` modification in the previous section calls a `notify()` method. Obviously, we need to implement that. Here is an implementation to add to `TwitterMonitor`:

```
private void notify(String friend) {
  Notification note=new Notification(R.drawable.red_ball,
                                     "Tweet!",
                                     System.currentTimeMillis());
  Intent i=new Intent();

  i.setClassName("apt.tutorial.two",
                 "apt.tutorial.two.Patchy");
  i.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);

  PendingIntent pi=PendingIntent.getActivity(this, 0, i,
                                             0);

  note.setLatestEventInfo(this, "Tweet!",
                          friend+" updated their Twitter status",
                          pi);

  mgr.notify(NOTIFY_ME_ID, note);
}
```

Note that you will need to choose an appropriate icon and put it in `TMonitor/res/drawable` (in this case, it shows a PNG named `red_ball`).

Recompile and rebuild both projects. Then, start up `Patchy`, click over to the Friends tab, check off a friend or two, and choose the Set Best Friends option menu item. Then, when the friends update their status, you will receive a "Tweet!" notification in your status bar.

Note that `Patchy` and `TwitterMonitor` are somewhat tied together, in that `TwitterMonitor` will close down after `Patchy` calls `unbindService()`, since there are no more clients. Moreover, our BFF list is tied to an account, which is removed when `Patchy` closes down. As such, right now, when you test `Patchy`, you will need to keep `Patchy` running in order to receive the notification.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Allow each BFF to have its own notification characteristics (e.g., icon, ringtone). Store the BFF list and its notification settings in a database. Change the AIDL to simply alert the service of BFF changes. Have the service use the database for the BFF roster and configure the notifications based on the individual BFF settings.
- When the user goes into `Patchy` via the notification, ensure the status from the specific BFF is shown in the list.
- Support multiple BFF status updates via the notification count mechanism.
- Arrange to do the initial population of the friends list in a background thread, either proactively updated periodically (e.g., via `TwitterMonitor`) or by a refresh menu action.
- Save the BFF roster between `Patchy` runs, either via a database table or via a local file (e.g., JSON dump of friend screen names).

Tweets On Location

In this tutorial, we will integrate location tracking, such that we can optionally embed our location in our status updates following the Twittervision convention.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `22-Notifications` edition of `Patchy` to use as a starting point.

Step #1: Get the LocationManager

First, we need to arrange to have access to Android location services through the Location Manager.

This requires a permission, either `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION`. Since the emulator simulates GPS, which needs `ACCESS_FINE_LOCATION`, modify `Patchy/AndroidManifest.xml` to add this permission:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="apt.tutorial.two"
```

```
      android:versionCode="1"
      android:versionName="1.0">
  <uses-permission android:name="android.permission.INTERNET" />
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <application android:label="@string/app_name">
        <activity android:name=".Patchy"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".EditPreferences">
        </activity>
    </application>
</manifest>
```

Then, add some imports to `Patchy` for location-related classes that we will need:

```
import android.location.Location;
import android.location.LocationListener;
import android.location.LocationManager;
```

Next, add a `LocationManager` data member named `locMgr` in `Patchy`, then initialize it in `onCreate()`:

```
locMgr=(LocationManager)getSystemService(Context.LOCATION_SERVICE);
```

At this point, we can start using the `LocationManager` for getting the location to embed in the status update.

Step #2: Register for Location Updates

Next, we need to do something to cause Android to actually activate GPS and get fixes. Just having access to `LocationManager` is insufficient. The simplest answer is to register for location updates, even if we will use another way to get our current fix, as you will see later in this tutorial.

So, add the following statement to `onCreate()` in `Patchy`, after you have initialized `locMgr` as shown in the preceding section:

```
locMgr.requestLocationUpdates(LocationManager.GPS_PROVIDER,
                              10000,
                              10000.0f,
                              onLocationChange);
```

This refers to an `onLocationChange` object, which is a `LocationListener`. We do not truly care about the updates (though, in principle, our application could do something with them). Hence, for this tutorial's purpose, all you need is a stub implementation of `LocationListener`. Add the following to `Patchy`:

```
LocationListener onLocationChange=new LocationListener() {
  public void onLocationChanged(Location location) {
    // required for interface, not used
  }

  public void onProviderDisabled(String provider) {
    // required for interface, not used
  }

  public void onProviderEnabled(String provider) {
    // required for interface, not used
  }

  public void onStatusChanged(String provider, int status,
                              Bundle extras) {
    // required for interface, not used
  }
};
```

Finally, since we registered for location updates in `onCreate()`, we should remove that request in `onDestroy()`. Modify `onDestroy()` in `Patchy` to look like this:

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

  locMgr.removeUpdates(onLocationChange);

  try {
    if (service!=null) {
      service.removeAccount(listener);
    }
  }
  catch (Throwable t) {
    Log.e("Patchy", "Exception in call to removeAccount()", t);
    goBlooey(t);
  }
```

```
  unbindService(svcConn);
}
```

Step #3: Add "Insert Location" Menu

Now, we need to give the user a way to inject the current location into the status update being written. The simplest way to do that is by adding a menu item to `Patchy/res/menu/option.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/prefs"
    android:title="Settings"
    android:icon="@android:drawable/ic_menu_preferences"
  />
  <item android:id="@+id/bff"
    android:title="Set Best Friends"
    android:icon="@drawable/friends"
  />
  <item android:id="@+id/location"
    android:title="Insert Location"
    android:icon="@android:drawable/ic_menu_compass"
  />
</menu>
```

If you recompile and reinstall the application, you will see the new menu option, even though it does not do anything just yet:

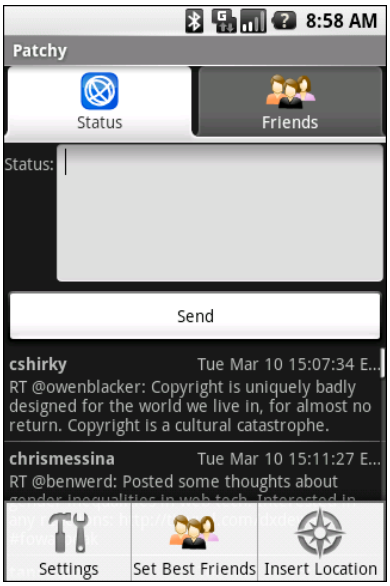


Figure 39. The Insert Location menu item in Patchy

Step #4: Insert the Last Known Location

Finally, we need to do something to add the location to our status message.

Change `onOptionsItemSelected()` in `Patchy` to look like this:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.prefs) {
    startActivity(new Intent(this, EditPreferences.class));

    return(true);
  }
  else if (item.getItemId()==R.id.bff) {
    try {
      List<String> bff=new ArrayList<String>();

      for (int i=0;i<friends.size();i++) {
        if (friendsList.isItemChecked(i)) {
          bff.add(friends.get(i));
        }
      }

      service.setBestFriends(listener, bff);
    }
```

```
    catch (Throwable t) {
      Log.e("Patchy",
            "Exception in onOptionsItemSelected()", t);
      goBlooey(t);
    }

    return(true);
  }
  else if (item.getItemId()==R.id.location) {
    insertLocation();

    return(true);
  }

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

Then, add in the following implementation of `insertLocation()` in `Patchy`:

```
private void insertLocation() {
  Location loc=locMgr.getLastKnownLocation(LocationManager.GPS_PROVIDER);

  if (loc==null) {
    Toast
      .makeText(this,
                "No location available",
                Toast.LENGTH_SHORT)
      .show();
  }
  else {
    StringBuffer buf=new StringBuffer(status
                                      .getText()
                                      .toString());

    buf.append(" L:");
    buf.append(String.valueOf(loc.getLatitude()));
    buf.append(",");
    buf.append(String.valueOf(loc.getLongitude()));

    status.setText(buf.toString());
  }
}
```

Now, if you recompile and reinstall `Patchy`, and if you use DDMS to supply a fake location (assuming you are running this on an emulator), the location will be appended to the end of the status update when you choose the Insert Location option menu item:

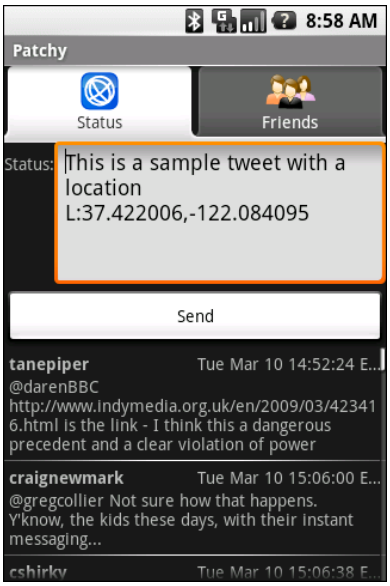


Figure 40. The results from the Insert Location menu item in Patchy

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Allow the user to choose the `LocationProvider` via a set of preferences that specify the attributes of a `Criteria` object, then using that `Criteria` object to get the best-matching `LocationProvider`.
- Use `onKeyDown()` to add "hot-key" support, such that some key (e.g., <Alt>-<X>) will inject the location, instead of having to use the option menu.

TUTORIAL 24

Here a Tweet, There a Tweet

In this tutorial, we will integrate maps, so we can plot the location for status updates that provide such information. Along the way, we will add support to view the public timeline to `Patchy`.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `23-Location` edition of `Patchy` to use as a starting point.

Step #1: Register for a Map API Key

First, you need to register for an API key to use with the mapping services and set it up in your development environment with your debug certificate. Full instructions for doing this can be found on the Android developer site.

Step #2: Create a Basic MapActivity

Next, we need to create a stub `MapActivity` implementation that we can then use in `Patchy`.

Create `Patchy/res/layout/status_map.xml` with the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <com.google.android.maps.MapView android:id="@+id/map"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:apiKey="00yHj0k7_7vx5UPXcC6nXvDJtLej09bmZHFmAVQ"
    android:clickable="true" />
  <LinearLayout android:id="@+id/zoom"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:layout_alignParentLeft="true" />
</RelativeLayout>
```

Note that you will need to substitute your API key for the one shown in `android:apiKey` in the above code listing.

Next, create `Patchy/src/apt/tutorial/two/StatusMap.java` with the following content:

```
package apt.tutorial.two;

import android.app.Activity;
import android.os.Bundle;
import android.view.ViewGroup;
import com.google.android.maps.MapActivity;
import com.google.android.maps.MapController;
import com.google.android.maps.MapView;

public class StatusMap extends MapActivity {
  private MapView map=null;

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

    map=(MapView)findViewById(R.id.map);

    map.getController().setZoom(17);
  }

  @Override
  protected boolean isRouteDisplayed() {
    return(false);
  }
}
```

All we do here is create a map and set the initial zoom level.

Then, we need to make two changes to our `AndroidManifest.xml` file: we need to add the `StatusMap` activity, and we need to indicate that we are using the mapping services library. Alter `Patchy/AndroidManifest.xml` to look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="apt.tutorial.two"
      android:versionCode="1"
      android:versionName="1.0">
  <uses-permission android:name="android.permission.INTERNET" />
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <application android:label="@string/app_name">
        <uses-library android:name="com.google.android.maps" />
        <activity android:name=".Patchy"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".EditPreferences">
        </activity>
        <activity android:name=".StatusMap">
        </activity>
    </application>
</manifest>
```

Step #3: Launch the MapActivity on Location-Bearing Status Click

Now, we need to tie `Patchy` and `StatusMap` together, such that a click on a status that has an embedded location triggers the display of the map.

To do this, first we need to get control when a status is clicked. Right now, the `onCreate()` in `Patchy` has a line that looks like:

```
((ListView)findViewById(R.id.timeline)).setAdapter(adapter);
```

Remove that line and replace it with:

```
ListView list=(ListView)findViewById(R.id.timeline);

list.setAdapter(adapter);
list.setOnItemClickListener(onStatusClick);
```

That gives control on a click to an `onStatusClick` object. In there, we need to get the `TimelineEntry` corresponding with the click, scan the status to find the location (if any), and pass that information to `StatusMap` via `startActivity()`.

Add the following `onStatusClick` implementation to `Patchy`:

```
private AdapterView.OnItemClickListener onStatusClick=
                  new AdapterView.OnItemClickListener() {
  public void onItemClick(AdapterView<?> parent, View view,
                          int position, long id) {
    TimelineEntry entry=timeline.get(position);
    Matcher r=regexLocation.matcher(entry.status);

    if (r.find()) {
      double latitude=Double.valueOf(r.group(1));
      double longitude=Double.valueOf(r.group(4));

      Intent i=new Intent(Patchy.this, StatusMap.class);

      i.putExtra(LATITUDE, latitude);
      i.putExtra(LONGITUDE, longitude);
      i.putExtra(STATUS_TEXT, entry.status);

      startActivity(i);
    }
  }
};
```

This requires a few constants to be added to `Patchy` as well:

```
public static final String LATITUDE="apt.tutorial.latitude";
public static final String LONGITUDE="apt.tutorial.longitude";
public static final String STATUS_TEXT="apt.tutorial.statusText";
```

We also need the `Pattern` that defines the regular expression we are searching for:

```
private Pattern regexLocation=Pattern.compile("L\\:((\\-)?[0-9]+(\\.[0-9]+)?)\\,
((\\-)?[0-9]+(\\.[0-9]+)?)");
```

Now, when we click on a location-bearing status, `StatusMap` will open. However, we need to add some logic to `StatusMap` to unpack the latitude and longitude and center the map on that position.

Change `StatusMap` to look like the following:

```
package apt.tutorial.two;

import android.app.Activity;
import android.os.Bundle;
import android.view.ViewGroup;
import com.google.android.maps.GeoPoint;
import com.google.android.maps.MapActivity;
import com.google.android.maps.MapController;
import com.google.android.maps.MapView;

public class StatusMap extends MapActivity {
  private MapView map=null;

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

    map=(MapView)findViewById(R.id.map);

    map.getController().setZoom(17);

    double lat=getIntent().getDoubleExtra(Patchy.LATITUDE, 0);
    double lon=getIntent().getDoubleExtra(Patchy.LONGITUDE, 0);

    GeoPoint status=new GeoPoint((int)(lat*1000000.0),
                                 (int)(lon*1000000.0));

    map.getController().setCenter(status);
  }

  @Override
  protected boolean isRouteDisplayed() {
    return(false);
  }
}
```

Now, if you rebuild and reinstall `Patchy`, and you click on a status that has a location (`L:nnn.nn,nnn.nn` – you may need somebody to update their status with a location for you!), it will open the map on that location.

Step #4: Add Zoom Controls

Next, it would be nice if the user could control the zoom levels. We have set aside space in the layout for the zoom controls, in the lower-left corner. However, we need some code to tell the `MapView` to use that space for the zoom controls.

To do that, add the following lines to `onCreate()` in `StatusMap`:

```
ViewGroup zoom=(ViewGroup)findViewById(R.id.zoom);

zoom.addView(map.getZoomControls());
```

Step #5: Show the Location Via a Pin

Finally, it would be good to put a pin on the map at the spot identified by the status. Not only will this help the user find the location again after panning around the map, but we can also have the pin respond to a click by displaying the text of the status update.

First, find yourself a suitable pin image, probably on the order of 32x32 pixels. In this code, we assume that the pin image is called `marker`.

Next, add the following lines to `onCreate()` in `StatusMap`:

```
String statusText=getIntent().getStringExtra(Patchy.STATUS_TEXT);
Drawable marker=getResources().getDrawable(R.drawable.marker);

marker.setBounds(0, 0, marker.getIntrinsicWidth(),
                       marker.getIntrinsicHeight());

map.getOverlays().add(new StatusOverlay(marker, status,
                                        statusText));
```

This requires an implementation of `StatusOverlay` – add the following inner class to `StatusMap`:

```
private class StatusOverlay extends ItemizedOverlay<OverlayItem> {
  private OverlayItem item=null;
  private Drawable marker=null;
```

```
  public StatusOverlay(Drawable marker, GeoPoint status,
                       String statusText) {
    super(marker);
    this.marker=marker;

    item=new OverlayItem(status, "Tweet!", statusText);

    populate();
  }

  @Override
  protected OverlayItem createItem(int i) {
    return(item);
  }

  @Override
  public void draw(Canvas canvas, MapView mapView,
                   boolean shadow) {
    super.draw(canvas, mapView, shadow);

    boundCenterBottom(marker);
  }

  @Override
  protected boolean onTap(int i) {
    Toast.makeText(StatusMap.this,
                   item.getSnippet(),
                   Toast.LENGTH_SHORT).show();

    return(true);
  }

  @Override
  public int size() {
    return(1);
  }
}
```

Now, when you view the map, you will see the pin at the spot of the location:

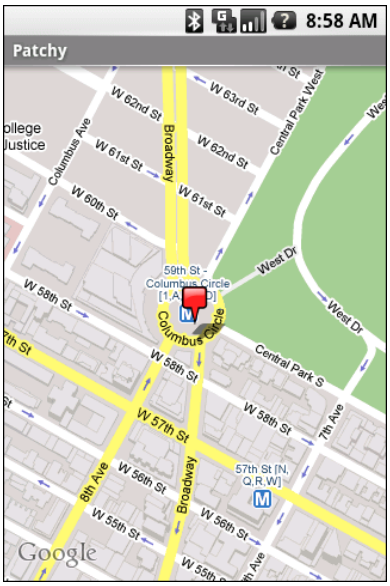


Figure 41. The StatusMap, showing a location in New York City

Clicking on the pin will display a `Toast` with the text of the status update itself:

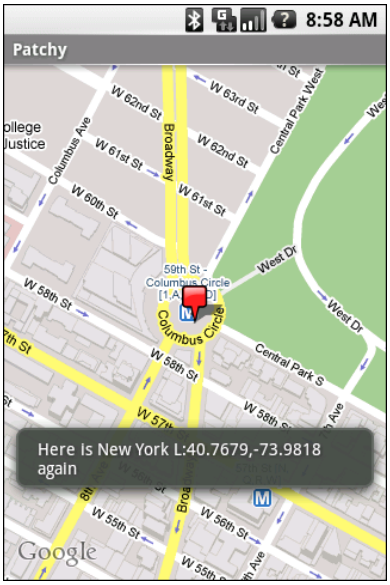


Figure 42. The StatusMap, showing a Toast of the status update associated with the location

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Support other map perspectives, such as street view or terrain mode.
- Allow the user to control the initial zoom level via a preference.
- Enable the compass rose, via `MyLocationOverlay`.

Media

In this tutorial, we will integrate a video clip, to serve as a placeholder for some sort of future "helpcast" to assist users with `Patchy`.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `24-Maps` edition of `Patchy` to use as a starting point.

Step #1: Obtain and Install a Video Clip

First, we need a suitable video clip to serve as our helpcast placeholder. A 320x240 MP4 video file should do nicely. Please, though, respect creators copyright – consider using a Creative Commons-licensed clip, such as one of these.

For something the size of your typical video clip, you should store it in an SD card, since space in on-board flash memory is at a premium. If you are using an emulator, and you do not have an SD card image set up for it, do the following:

1. Use the `mksdcard` utility, provided in the `tools/` directory of your SDK installation, to create an SD card image (e.g., `mksdcard 512M`

sdcard.img). Make sure the image file is big enough to hold your video clip.

2. Launch the emulator with the -sdcard switch, to point the emulator at the SD card image you created above (e.g., emulator -sdcard path/to/my/sdcard.img)

Once you have real or emulated SD card storage ready, we need to create a .Patchy folder on there and upload the MP4 file there as helpcast.mp4. To do that, either use Eclipse's file-browsing tools, or execute adb shell "mkdir /sdcard/.Patchy" from the command line, followed by adb push path/to/video.mp4 /sdcard/.Patchy/helpcast.mp4.

Step #2: Create the Stub Helpcast Activity

Now, we need to add an activity that will use the VideoView widget to display our video clip.

First, add the following layout as Patchy/res/layout/helpcast.xml:

```
<?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"
    >
  <VideoView
    android:id="@+id/video"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
    />
</LinearLayout>
```

Then, create Patchy/src/apt/tutorial/two/HelpCast.java with the following code:

```
package apt.tutorial.two;

import android.app.Activity;
import android.graphics.PixelFormat;
import android.os.Bundle;
import android.view.View;
import android.widget.MediaController;
```

```
import android.widget.VideoView;
import java.io.File;

public class HelpCast extends Activity {
  private VideoView video;
  private MediaController ctlr;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    getWindow().setFormat(PixelFormat.TRANSLUCENT);
    setContentView(R.layout.helpcast);

    File clip=new File("/sdcard/.Patchy/helpcast.mp4");

    if (clip.exists()) {
      video=(VideoView)findViewById(R.id.video);
      video.setVideoPath(clip.getAbsolutePath());

      ctlr=new MediaController(this);
      ctlr.setMediaPlayer(video);
      video.setMediaController(ctlr);
      video.requestFocus();
    }
  }
}
```

Here, we simply configure the `VideoView`, including the `MediaController` to give us buttons to control playback of the video.

And, as always, when we add an activity to our project, we need to add it to `AndroidManifest.xml`, so add the following `<activity>` element alongside the others:

```
<activity android:name=".HelpCast">
</activity>
```

Step #3: Launch the Helpcast from the Menu

Finally, we need to integrate `HelpCast` into `Patchy` overall, so users can display the help video. As we have done in previous tutorials, we will accomplish this by extending the option menu with another menu item.

First, add the following `<item>` to `Patchy/res/menu/option.xml`:

```
<item android:id="@+id/help"
  android:title="Help"
  android:icon="@android:drawable/ic_menu_help"
/>
```

Then, update `onOptionsItemSelected()` in `Patchy` to launch `HelpCast` when our item is selected:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.prefs) {
    startActivity(new Intent(this, EditPreferences.class));

    return(true);
  }
  else if (item.getItemId()==R.id.bff) {
    try {
      List<String> bff=new ArrayList<String>();

      for (int i=0;i<friends.size();i++) {
        if (friendsList.isItemChecked(i)) {
          bff.add(friends.get(i));
        }
      }

      service.setBestFriends(listener, bff);
    }
    catch (Throwable t) {
      Log.e("Patchy",
            "Exception in onOptionsItemSelected()", t);
      goBlooey(t);
    }

    return(true);
  }
  else if (item.getItemId()==R.id.location) {
    insertLocation();

    return(true);
  }
  else if (item.getItemId()==R.id.help) {
    startActivity(new Intent(this, HelpCast.class));

    return(true);
  }

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

With all that complete, we have our helpcast ready to go. Rebuild and reinstall the application, and you will see the new help menu item:

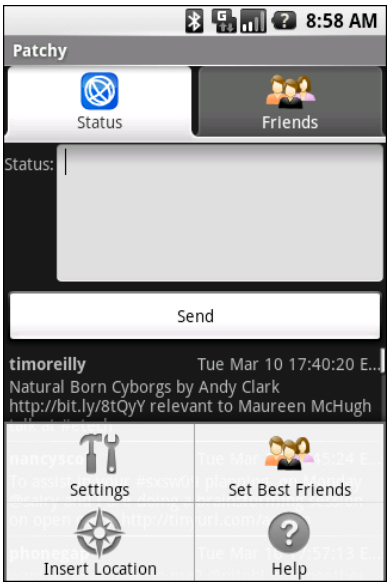


Figure 43. The new option menu item in Patchy

Clicking on it will bring up our "helpcast" video clip:

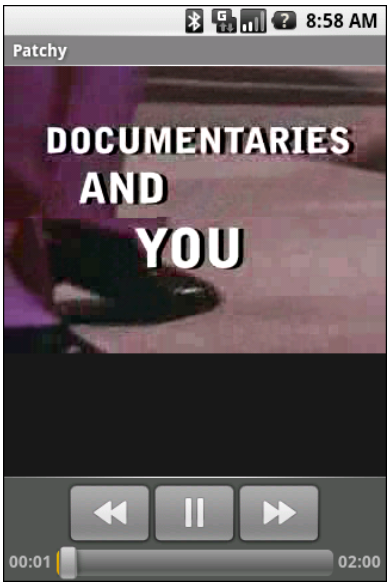


Figure 44. The "helpcast" placeholder

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Experiment with streaming video. Note that the requirements for Android streaming video, as of the 1.1 SDK release, are rather arcane and ill-documented.
- If you can find several media clips that work, put a `Spinner` on the `HelpcastActivity` to allow the user to switch between video clips.

Twitter? In a Browser???

In this tutorial, we will support clicking on links found in status updates, popping those up in the Browser application. We will also add a help screen that loads a local help HTML file, by integrating WebKit into `Patchy`.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `25-Media` edition of `Patchy` to use as a starting point.

Step #1: Add Auto-Linking

We can start by letting Android "do the heavy lifting for us" in making links in status updates clickable. That is merely a matter of adding `android:autoLink = "all"` to the `TextView` for the status text in `Patchy/res/layout/row.xml`. With a value of `"all"`, we are saying that we want all recognized patterns to be converted to links: Web URLs, email addresses, phone numbers, etc.

If you make this change, then recompile and reinstall the application, you will notice two things:

1. All of the clickable items show up in blue underlined text, and when clicked they pop up the appropriate application (e.g., clicking a Web URL brings up the Browser application).
2. Clicking on a row with a location, as we introduced in a previous tutorial, does not spawn our `StatusMap`.

Step #2: Draft and Package the Help HTML

Next, we need some placeholder HTML to serve as our help prose. This does not need to be terribly fancy – in fact, simpler HTML works better, because it loads faster.

So, write a Web page that will serve as the placeholder for the `Patchy` help. The key is where you put the page: create an `assets/` directory in your project and store it as `help.html` in there. That will line up with the URL we will use in the next section to reference that help file.

Step #3: Create a Help Activity

Now, we can create a help activity class that will load our Web page and do some other useful things.

First, create `Patchy/res/layout/help.xml` with the following content:

```
<?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"
  >
  <WebView android:id="@+id/webkit"
    android:layout_width="fill_parent"
    android:layout_height="0px"
    android:layout_weight="1"
  />
</LinearLayout>
```

Then, create `Patchy/src/apt/tutorial/two/HelpPage.java` with the following code:

```
package apt.tutorial.two;

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

public class HelpPage extends Activity {
  private WebView browser;

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

    browser=(WebView)findViewById(R.id.webkit);
    browser.loadUrl("file:///android_asset/help.html");
  }
}
```

Note how we use `file:///android_asset/help.html` as the URL syntax to reach into our APK's assets to load the desired Web page.

Finally, as normal, we need to add another `<activity>` element to our `AndroidManifest.xml` file:

```
<activity android:name=".HelpPage">
</activity>
```

Step #4: Splice In the Help Activity

Right now, the option menu in `Patchy` for help launches the `HelpCast` activity. We want to change that so `Patchy` launches `HelpPage` instead, then augment `HelpPage` to display `HelpCast` on demand.

Making the change to display `HelpPage` is easy – just replace the `HelpCast` reference with `HelpPage` in `onOptionsItemSelected()`:

```
startActivity(new Intent(this, HelpPage.class));
```

At this point, if you rebuild and reinstall the application, then click on the help menu item, you will see your help page in a full-screen browser.

To get to `HelpCast`, let us add a `Button` to the `HelpPage` layout and wire it up so, when clicked, the `Button` launches `HelpCast`.

First, add a suitable `Button` to `Patchy/res/layout/help.xml`:

```
<?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"
  >
  <WebView android:id="@+id/webkit"
    android:layout_width="fill_parent"
    android:layout_height="0px"
    android:layout_weight="1"
  />
  <Button android:id="@+id/helpcast"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="View Helpcast"
  />
</LinearLayout>
```

Then, add logic to `HelpCast` to find the `Button` and set the on-click listener to an object that will call `startActivity()` on `HelpCast`:

```
package apt.tutorial.two;

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

public class HelpPage extends Activity {
  private WebView browser;

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

    browser=(WebView)findViewById(R.id.webkit);
    browser.loadUrl("file:///android_asset/help.html");

    Button cast=(Button)findViewById(R.id.helpcast);

    cast.setOnClickListener(onCast);
  }
```

```
  private View.OnClickListener onCast=new View.OnClickListener() {
    public void onClick(View v) {
      startActivity(new Intent(HelpPage.this, HelpCast.class));
    }
  };
}
```

Now, if you recompile and reinstall the application, clicking the help menu item brings up `HelpPage` with both help content and the "View HelpCast" button:

Figure 45. The HelpPage activity

Clicking the button, in turn, will display the "helpcast" video clip.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Support multiple pages of help text, by using `WebViewClient` and `shouldOverrideUrlLoading()`.
- Experiment with adding images or CSS stylesheets to the help page.

- Support the notion of the help being updated separately from the APK. Search some URL for help updates, downloading them in the background, and using them if available, falling back to the in-APK edition if there are no such updates.
- Instead of detecting locations when statuses are clicked upon, detect locations in the constructor for `TimelineEntry`. Then, add a "Map" button to the row layout, set to `GONE` if there is no location, or `VISIBLE` if there is one. Then, arrange to have clicking the Map button bring up the `StatusMap`.

High-Priced Help

In this tutorial, we will extend our help system to embed user details, such as their Twitter screen name, in the help text itself on the fly, by way of injecting Java objects into the `WebView` Javascript engine.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `26-WebKit` edition of `Patchy` to use as a starting point.

Step #1: Enable Javascript

By default, `WebView` widgets have Javascript disabled. Since our current help page does not use Javascript, that was not a problem in the previous tutorial. Now, however, we are looking to add a script to the page, so we need to enable Javascript in our `WebView`.

Add this line to `onCreate()` in `HelpPage`:

```
browser.getSettings().setJavaScriptEnabled(true);
```

That is all you need to enable basic Javascript operation.

Step #2: Create the Java Object to Inject

Next, we need to create a Java object from a public class with a public method that we can inject into the Javascript environment of our `WebView` that will give us access to the user's Twitter user name.

To that end, add the following inner class to `HelpPage`:

```
public class CustomHelp {
  SharedPreferences prefs=null;

  CustomHelp() {
    prefs=PreferenceManager
              .getDefaultSharedPreferences(HelpPage.this);
  }

  public String getUserName() {
    return(prefs.getString("user", "<no account>"));
  }
}
```

Step #3: Inject the Java Object

Just because we created the `CustomHelp` inner class does not mean Javascript has access to it. Instead, we need to give the `WebView` an instance of `CustomHelp` and associate it with a name that will be used to make the `CustomHelp` instance look like a global variable in Javascript.

Add the following lines to `onCreate()` in `HelpPage`:

```
browser.addJavascriptInterface(new CustomHelp(),
                               "customHelp");
```

The entirety of `onCreate()` in `HelpPage` should now look like this:

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

  browser=(WebView)findViewById(R.id.webkit);
  browser.getSettings().setJavaScriptEnabled(true);
  browser.addJavascriptInterface(new CustomHelp(),
```

```
                                  "customHelp");
  browser.loadUrl("file:///android_asset/help.html");

  Button cast=(Button)findViewById(R.id.helpcast);

  cast.setOnClickListener(onCast);
}
```

In particular, we need to enable Javascript in the `WebView` and inject our `CustomHelp` object (under the name `customHelp`) before we load our help Web page.

Step #4: Leverage the Java Object from Javascript

With all that done, we can take advantage of the `customHelp` object in our Web page.

Somewhere on your page, add a `<div>` or `<span>` with an id of `userName`, such as:

```
<p>Your Twitter account name is:
<span id="userName"><i>unknown</i></span>!</p>
```

Then, add a global Javascript function that will replace the default contents of the `userName` element with the value obtained from `customHelp`, such as:

```
<script language="javascript">
  function updateUserName() {
    document
      .getElementById("userName")
      .innerHTML=customHelp.getUserName();
  }
</script>
```

Finally, add an `onLoad` attribute to your `<body>` element to trigger calling the global Javascript function, such as:

```
<body onLoad="updateUserName()">
```

If you do all of that, then rebuild and reinstall the application, the `Patchy` help page should show your Twitter user name where you placed it on the page:

Figure 46. The HelpPage activity, showing the results of our injected Java object

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Embed a link in the help HTML that will pop open the Browser application on the user's own Twitter page
- Have the help page optionally show the user's current location as an example of what would be injected into a status message. On a location update, call a global Javascript function in the page, so it can update the help to match the current location.

PART III – Advanced Tutorials

Even Fancier Lists

In this tutorial, we will make more changes to the `Patchy` list of status updates, including adding a button to request a refresh of the statuses and changing the selector from the normal highlight bar to a highlight box.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `27-AdvWebKit` edition of `Patchy` to use as a starting point.

Step #1: Exposing poll()

We want to add a button to our UI to manually request a timeline update, rather than always wait a minute. To do that, we need to make the logic behind `poll()` in `TwitterMonitor` available to `Patchy` via AIDL.

First, since `poll()` may now be called from different threads, add the synchronized keyword to the method declaration, so only one `poll()` can be done at a time:

```
synchronized private void poll(Account l) {
```

Next, we need to add an API to our `ITwitterMonitor` AIDL, such that the client can call upon the `poll()` functionality. To distinguish the API from the internal implementation, let us call the API `updateTimeline()`. So, add a simple `updateTimeline()` declaration to the AIDL:

```
package apt.tutorial;

import apt.tutorial.ITwitterListener;

interface ITwitterMonitor {
  void registerAccount(String user, String password,
                       ITwitterListener callback);
  void removeAccount(ITwitterListener callback);
  void setBestFriends(ITwitterListener callback,
                      in List<String> bff);
  void updateTimeline();
}
```

Then, we need to implement `updateTimeline()` to poll our accounts. Since we block the caller for as long as this takes, we need to do the work in a background thread. So, update our `ITwitterMonitor.Stub` to look like:

```
private final ITwitterMonitor.Stub binder=new ITwitterMonitor.Stub() {
  public void registerAccount(String user, String password,
                              ITwitterListener callback) {
    registerAccountImpl(user, password, callback);
  }

  public void removeAccount(ITwitterListener callback) {
    removeAccountImpl(callback);
  }

  public void setBestFriends(ITwitterListener callback,
                             List<String> newBff) {
    bff.clear();
    bff.addAll(newBff);
  }

  public void updateTimeline() {
    new Thread(new Runnable() {
      public void run() {
        for (Account l : accounts.values()) {
          poll(l);
        }
      }
    }).start();
  }
};
```

At this point, you should be able to rebuild and reinstall the `TMonitor` project onto your device or emulator.

Also, copy the modified AIDL file to the `Patchy` project, so both projects are synchronized.

Step #2: Adding a List Header

Now that we can request a timeline update, we need to modify `Patchy` to use it. Along the way, we will demonstrate adding a list header widget.

First, implement a `buildHeader()` method in `Patchy` that creates a `Button` that, when clicked, will call our new `updateTimeline()` API:

```
private View buildHeader() {
  Button btn=new Button(this);

  btn.setText("Update Me!");
  btn.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
      try {
        service.updateTimeline();
      }
      catch (Throwable t) {
        Log.e("Patchy", "Exception in update-me button", t);
        goBlooey(t);
      }
    }
  });

  return(btn);
}
```

To inject that button into the list at the top, simply add the following line to `onCreate()` in `Patchy`, after you have retrieved the list widget but before you have called its `setAdapter()`:

```
list.addHeaderView(buildHeader());
```

At this point, you should be able to recompile and reinstall `Patchy`. When running it, once the list is populated, you will see the "Update Me!" button.

This button is part of the list, so it will scroll off-screen as the list is scrolled. If you click the button, it will update your timeline.

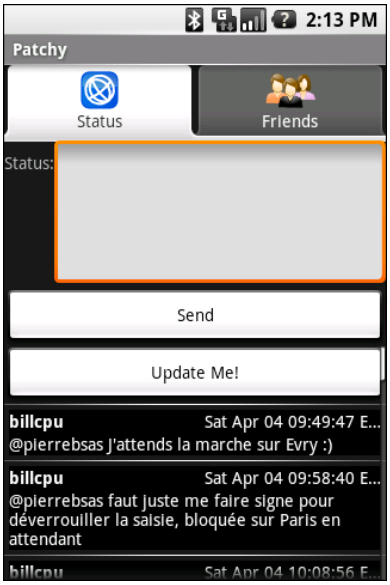


Figure 47. Patchy, now with an "Update Me!" button

Step #3: Boxing the Selected Status

To make `Patchy` even more distinctive, let us replace the normal selector bar with a red-box highlight. When traversing the list with the D-pad, you will see a red box outline the selected status, rather than have it filled with an orange bar.

First, we need to adjust our row layout XML. We want to apply a red background to our outer-most `LinearLayout`, but only when it is selected. However, by default, the immediate children of the `LinearLayout` have transparent backgrounds, meaning everything would be red when selected, not just our `4px` padding box. So, add `android:background = "#FF000000"` attributes to the children of the root `LinearLayout`:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
```

```
  android:orientation="vertical"
  android:padding="4px"
  >
  <LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:layout_weight="1"
    android:gravity="center_vertical"
    android:background="#FF000000"
    >
    <TextView android:id="@+id/friend"
      android:layout_width="0px"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:gravity="left"
      android:textStyle="bold"
      android:singleLine="true"
      android:ellipsize="end"
    />
    <TextView android:id="@+id/created_at"
      android:layout_width="0px"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:gravity="right"
      android:singleLine="true"
      android:ellipsize="end"
    />
  </LinearLayout>
  <TextView android:id="@+id/status"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:gravity="center_vertical"
    android:singleLine="false"
    android:autoLink="all"
    android:background="#FF000000"
  />
</LinearLayout>
```

Next, we need to tell Android to get rid of the normal list selector. To do this, add `android:listSelector = "#00000000"` to the timeline `ListView` in `Patchy/res/layout/main.xml`:

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

```
    <TabWidget android:id="@android:id/tabs"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
    />
    <FrameLayout android:id="@android:id/tabcontent"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:paddingTop="62px"
      >
      <TableLayout android:id="@+id/status_tab"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:stretchColumns="1"
        >
        <TableRow>
          <TextView android:text="Status:" />
          <EditText android:id="@+id/status"
            android:singleLine="false"
            android:gravity="top"
            android:lines="5"
            android:scrollHorizontally="false"
            android:maxLines="5"
            android:maxWidth="200sp"
          />
        </TableRow>
        <Button android:id="@+id/send"
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="Send"
        />
        <ListView android:id="@+id/timeline"
          android:layout_width="fill_parent"
          android:layout_height="fill_parent"
          android:listSelector="#00000000"
        />
      </TableLayout>
      <ListView android:id="@+id/friends"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
      />
    </FrameLayout>
  </TabHost>
</LinearLayout>
```

Now, we need to get control when list items are selected. Add the following statement in `onCreate()` in `Patchy`, sometime after we retrieve the `list` widget:

```
list.setOnItemSelectedListener(selectionListener);
```

That statement, in turn, relies upon a `selectionListener` object, which we must also add to `Patchy`:

```
AdapterView.OnItemSelectedListener selectionListener=
  new AdapterView.OnItemSelectedListener() {
  View lastRow=null;

  public void onItemSelected(AdapterView<?> parent,
                             View view, int position,
                             long id) {
    if (lastRow!=null) {
      lastRow.setBackgroundColor(0x00000000);
    }

    view.setBackgroundColor(0xFFFF0000);
    lastRow=view;
  }

  public void onNothingSelected(AdapterView<?> parent) {
    if (lastRow!=null) {
      lastRow.setBackgroundColor(0x00000000);
      lastRow=null;
    }
  }
};
```

Here, we simply toggle the last row to have a normal transparent background and the newly-selected row to have a red background (`0xFFFF0000`). However, we need to add a declaration for `lastRow` to our data members:

```
private View lastRow=null;
```

If you rebuild and reinstall `Patchy` at this point, and use the D-pad to scroll down into the list, you will see our red box instead of the orange bar. However, you will also see that the text fades when selected, making it difficult to read.

That is because `TextView` (and subclasses) have more than one color. They have their normal color, plus a selected color. The idea is that, when a `TextView` is part of a selected row in a list, the selected color is automatically applied, so it provides appropriate contrast to the selection bar. We got rid of the selection bar, and so now our contrast is incorrect.

This is easily fixable, however.

First, add a static `ColorStateList` named `allWhite`, which sets all colors for all possible states to be plain white:

```
private static ColorStateList allWhite=ColorStateList.valueOf(0xFFFFFFFF);
```

Then, in `TimelineEntryWrapper`, apply `allWhite` to each of the `TextView` widgets:

```
TimelineEntryWrapper(View row) {
  this.row=row;

  getFriend().setTextColor(allWhite);
  getCreatedAt().setTextColor(allWhite);
  getStatus().setTextColor(allWhite);
}
```

Now, when you recompile and reinstall `Patchy`, you should get better selection behavior:

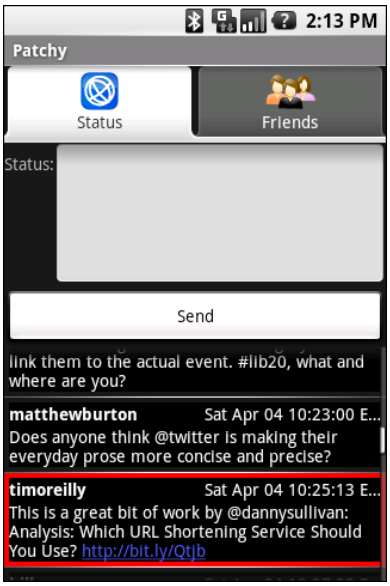


Figure 48. Patchy, complete with red-box selection highlighting

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Add a footer view showing the current number of people you are following (and, therefore, have updates in the timeline).
- Divide your timeline into BFF and regular people, each with a section heading in the `ListView`. This will likely involve many changes to the data model.

A List with a View

In this tutorial, we will create our own `View` class to represent the timeline status entry widget in `Patchy`. This `View` can then be used anywhere else we might use a `View`, including supporting XML attributes for use in layouts.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `28-AdvLists` edition of `Patchy` to use as a starting point.

Step #1: Create Stub StatusEntryView Class

For the purposes of this tutorial, we will have `StatusEntryView` be a subclass of `LinearLayout`. And, since we want to be able to obtain parameters from an XML layout, we need to implement the appropriate constructor.

So, create `Patchy/src/apt/tutorial/two/StatusEntryView.java` with the following starter code:

```
package apt.tutorial.two;

import android.content.Context;
import android.util.AttributeSet;
import android.widget.LinearLayout;
```

```
public class StatusEntryView extends LinearLayout {
  public StatusEntryView(final Context ctxt,
                         AttributeSet attrs) {
    super(ctxt, attrs);
  }
}
```

Step #2: Declare Our Attributes

Next, we should choose some attributes we want users of this widget to be able to set in their layouts. Two likely candidates are the text to use for the `TextView` and `Button` captions, so they can be suitably localized by the application.

With that in mind, create `Patchy/res/values/attrs.xml` with the following content:

```
<resources>
  <declare-styleable name="StatusEntryView">
    <attr name="labelCaption" format="string" />
    <attr name="buttonCaption" format="string" />
  </declare-styleable>
</resources>
```

Here, we indicate that we want two string attributes, `labelCaption` and `buttonCaption`.

Step #3: Define Our Layout

We also need to create a separate layout for our `StatusEntryView`, or intend to define its contents in Java code. For simplicity, if not necessarily efficiency, we will use a layout file.

Create `Patchy/res/layout/status_entry.xml` as follows:

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

```
  >
  <TableRow>
    <TextView android:id="@+id/label"
      android:text="Status:" />
    <EditText android:id="@+id/status"
      android:singleLine="false"
      android:gravity="top"
      android:lines="5"
      android:scrollHorizontally="false"
      android:maxLines="5"
      android:maxWidth="200sp"
    />
  </TableRow>
  <Button android:id="@+id/send"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Send"
  />
</TableLayout>
```

This is very similar to the layout we had in `main.xml`, just pulled out into its own file. We added an `android:id` attribute to the `TextView`, though, so we can get access to it from Java code.

Step #4: Finish the StatusEntryView Implementation

Of course, `StatusEntryView` lacks a bit of code. Specifically, we need to:

- Get our `labelCaption` and `statusCaption` values out of the layout XML
- Inflate our contents as defined in `status_entry.xml`
- Set the captions for the `TextView` and `Button`, if values were supplied to us to use
- Allow users of `StatusEntryView` to specify a listener that will get invoked when the `Button` is clicked
- Add logic such that when the `Button` is clicked, we call the supplied listener (if any)
- Allow users of `StatusEntryView` to get and set the actual status text in the `EditText` widget

That sounds like a lot, but it is not really all that much code. Here is a complete implementation of `StatusEntryView` that meets all of those requirements:

```
package apt.tutorial.two;

import android.app.Activity;
import android.content.Context;
import android.content.res.TypedArray;
import android.util.AttributeSet;
import android.view.View;
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.EditText;
import android.widget.LinearLayout;
import android.widget.TextView;

public class StatusEntryView extends LinearLayout {
  private String labelCaption=null;
  private String buttonCaption=null;
  private Button send=null;
  private View.OnClickListener onSend=null;
  private EditText status=null;

  public StatusEntryView(final Context ctxt,
                         AttributeSet attrs) {
    super(ctxt, attrs);

    TypedArray a=ctxt.obtainStyledAttributes(attrs,
                                             R.styleable.StatusEntryView,
                                             0, 0);

    labelCaption=a.getString(R.styleable.StatusEntryView_labelCaption);
    buttonCaption=a.getString(R.styleable.StatusEntryView_buttonCaption);

    a.recycle();
  }

  public void setOnSendListener(View.OnClickListener onSend) {
    this.onSend=onSend;
  }

  public String getText() {
    return(status.getText().toString());
  }

  public void setText(String text) {
    status.setText(text);
  }

  @Override
  protected void onFinishInflate() {
    super.onFinishInflate();
```

```
    ((Activity)getContext())
      .getLayoutInflater()
      .inflate(R.layout.status_entry, this);

    if (labelCaption!=null) {
      TextView label=(TextView)findViewById(R.id.label);

      label.setText(labelCaption);
    }

    send=(Button)findViewById(R.id.send);

    if (buttonCaption!=null) {
      send.setText(buttonCaption);
    }

    send.setOnClickListener(new View.OnClickListener() {
      public void onClick(View v) {
        if (onSend!=null) {
          onSend.onClick(StatusEntryView.this);
        }
      }
    });

    status=(EditText)findViewById(R.id.status);
  }
}
```

Make your class match this one, or otherwise make the required modifications.

Step #5: Use the StatusEntryView Class

Finally, we need to modify `Patchy` to make use of `StatusEntryView`.

The current layout for `Patchy` still has the individual widgets for the status entry, so replace them with a reference to `StatusEntryView`, making sure it appears above the `ListView` for the timeline:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tut="http://schemas.android.com/apk/res/apt.tutorial.two"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <TabHost android:id="@android:id/tabhost"
    android:layout_width="fill_parent"
```

```
    android:layout_height="fill_parent"
    >
    <TabWidget android:id="@android:id/tabs"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
    />
    <FrameLayout android:id="@android:id/tabcontent"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:paddingTop="62px"
      >
      <LinearLayout android:id="@+id/status_tab"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
        <apt.tutorial.two.StatusEntryView
          android:id="@+id/status_entry"
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          tut:labelCaption="Entry:"
          tut:buttonCaption="Tweet Me!"
        />
        <ListView android:id="@+id/timeline"
          android:layout_width="fill_parent"
          android:layout_height="fill_parent"
          android:listSelector="#00000000"
        />
      </LinearLayout>
      <ListView android:id="@+id/friends"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
      />
    </FrameLayout>
  </TabHost>
</LinearLayout>
```

Then, change the type of the status data member from `EditText` to `StatusEntryView`:

```
private StatusEntryView status=null;
```

That will also require some changes in `onCreate()`, to cast status to be a `StatusEntryView` and to set our `onSend` listener on the `StatusEntryView` rather than the `Button` as before. To do that, change:

```
status=(EditText)findViewById(R.id.status);

Button send=(Button)findViewById(R.id.send);

send.setOnClickListener(onSend);
```

to be:

```
status=(StatusEntryView)findViewById(R.id.status_entry);
status.setOnSendListener(onSend);
```

At that point, you should be able to rebuild and reinstall `Patchy`. When you run it, nothing will visibly be different...until the next tutorial, that is.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Try converting each timeline entry into a `TimelineEntryView`.
- Add other XML attributes (and corresponding accessor methods) of interest to `TimelineEntryView`.

Now Your Friends Seem Animated

Most of the time, we are reading status updates, not updating our own status. Hence, having the `StatusEntryView` always around takes up a lot of screen space. It would be nice to have it appear or disappear at the user's request.

This tutorial will cover that very process. We will give the user an option menu choice to show or hide the `StatusEntryView`. Initially, we will simply hide and show the widget without any animation. Then, we will use an `AlphaAnimation` to have the widget fade in or out.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `29-CustomViews` edition of `Patchy` to use as a starting point.

Step #1: Set Up the Option Menu

Let us start by giving the user the option to show and hide the `StatusEntryView` widget. That can be handled by just another entry in our

option menu...though we will need to do a little work to toggle the menu item from "show" to "hide" mode and back again.

First, add a `status_entry` menu option to `Patchy/res/menu/option.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/prefs"
    android:title="Settings"
    android:icon="@android:drawable/ic_menu_preferences"
  />
  <item android:id="@+id/status_entry"
    android:title="Hide Status Entry"
    android:icon="@drawable/status_hide"
  />
  <item android:id="@+id/bff"
    android:title="Set Best Friends"
    android:icon="@drawable/friends"
  />
  <item android:id="@+id/location"
    android:title="Insert Location"
    android:icon="@android:drawable/ic_menu_compass"
  />
  <item android:id="@+id/help"
    android:title="Help"
    android:icon="@android:drawable/ic_menu_help"
  />
</menu>
```

You will need to supply some PNG image to serve as the menu icon, preferably 32px tall.

The menu option is initially set up in "hide" mode. However, when the user pops up the menu, we want to change that menu option to "show" mode if the `StatusEntryView` is already hidden. To do this, we can use `onPrepareOptionsMenu()` – add the following method to `Patchy`:

```
@Override
public boolean onPrepareOptionsMenu(Menu menu) {
  MenuItem statusItem=menu.findItem(R.id.status_entry);

  if (status.getVisibility()==View.VISIBLE) {
    statusItem.setIcon(R.drawable.status_hide);
    statusItem.setTitle("Hide Status Entry");
  }
  else {
    statusItem.setIcon(R.drawable.status_show);
```

```
    statusItem.setTitle("Show Status Entry");
  }

  return(super.onPrepareOptionsMenu(menu));
}
```

Here, we see if the `StatusEntryView` is visible and set our menu item accordingly.

At this point, if you recompile and reinstall `Patchy`, you will see the new menu item, though it will not do anything just yet:

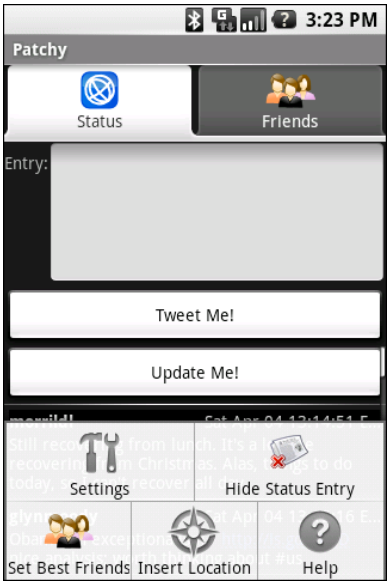


Figure 49. The "Hide Status Entry" menu option in Patchy

Step #2: Show and Hide the Widget

It would be nice, of course, if the newly-created menu item actually did something.

So, we need to add another case to `onOptionsItemSelected()` in `Patchy` to handle this new menu item:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.prefs) {
    startActivity(new Intent(this, EditPreferences.class));

    return(true);
  }
  else if (item.getItemId()==R.id.bff) {
    try {
      List<String> bff=new ArrayList<String>();

      for (int i=0;i<friends.size();i++) {
        if (friendsList.isItemChecked(i)) {
          bff.add(friends.get(i));
        }
      }

      service.setBestFriends(listener, bff);
    }
    catch (Throwable t) {
      Log.e("Patchy",
            "Exception in onOptionsItemSelected()", t);
      goBlooey(t);
    }

    return(true);
  }
  else if (item.getItemId()==R.id.location) {
    insertLocation();

    return(true);
  }
  else if (item.getItemId()==R.id.help) {
    startActivity(new Intent(this, HelpPage.class));

    return(true);
  }
  else if (item.getItemId()==R.id.status_entry) {
    toggleStatusEntry();

    return(true);
  }

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

Here, we merely delegate to a `toggleStatusEntry()` method, so add that method to `Patchy` as shown below:

```
private void toggleStatusEntry() {
  if (status.getVisibility()==View.VISIBLE) {
    status.setVisibility(View.GONE);
```

```
  }
  else {
    status.setVisibility(View.VISIBLE);
  }
}
```

All we do here is make the `StatusEntryView` hidden (`GONE`) or visible (`VISIBLE`), depending on its current state.

Now, when you hide the `StatusEntryView`, the timeline gets more room:

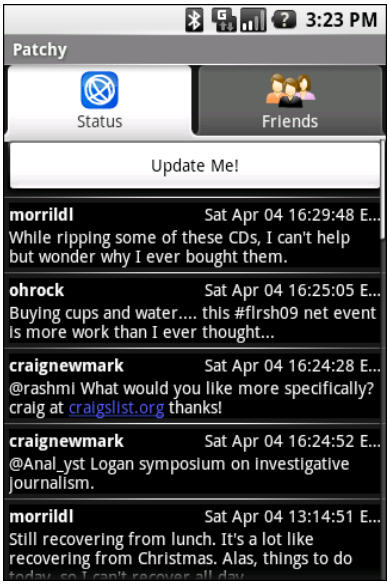


Figure 50. Patchy with a hidden StatusEntryView

Step #3: Fading In and Out

Of course, we have not yet used an animation, which is the point of this exercise.

First, we need to define our animations. In this case, we will use ones declared in animation XML files.

Create a `Patchy/res/anim` directory, then create `Patchy/res/anim/fade_out.xml` as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
  android:fromAlpha="1.0"
  android:toAlpha="0.0"
  android:duration="500" />
```

Also create `Patchy/res/anim/fade_in.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
  android:fromAlpha="0.0"
  android:toAlpha="1.0"
  android:duration="500"/>
```

These set up 500-millisecond alpha animations that fade out and fade in, respectively.

Next, we need to declare `Animation` objects as data members in `Patchy`, to hold these once we load them:

```
private Animation fadeOut=null;
private Animation fadeIn=null;
```

Then, we can use `AnimationUtils` to load our animation XML resources into the `Animation` objects. Add the following statements sometime late in the `onCreate()` method in `Patchy`:

```
fadeOut=AnimationUtils.loadAnimation(this, R.anim.fade_out);
fadeOut.setAnimationListener(fadeOutListener);

fadeIn=AnimationUtils.loadAnimation(this, R.anim.fade_in);
```

You will note that in addition to loading the animation resources, we attached an `AnimationListener` to the `fadeOut Animation`. That way, we get notified when the animation is done, so we can make the widget fully "gone". Add the following implementation of `fadeOutListener` to `Patchy`:

```
Animation.AnimationListener fadeOutListener=new Animation.AnimationListener() {
  public void onAnimationEnd(Animation animation) {
```

```
    status.setVisibility(View.GONE);
  }

  public void onAnimationRepeat(Animation animation) {
    // not needed
  }

  public void onAnimationStart(Animation animation) {
    // not needed
  }
};
```

Now, all that remains is to use `fadeOut` and `fadeIn` in our `toggleStatusEntry()` method:

```
private void toggleStatusEntry() {
  if (status.getVisibility()==View.VISIBLE) {
    status.startAnimation(fadeOut);
  }
  else {
    status.setVisibility(View.VISIBLE);
    status.startAnimation(fadeIn);
  }
}
```

At this point, you can rebuild and reinstall `Patchy`, and you will see your status entry area fade out when you hide it, then fade back in when you show it again.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Have the status entry widget slide out rather than fade out, either sliding out to the top or sliding out to one side.
- Use an `AnimationSet` to have the widget both fade and slide.

Friends, Drawn Together

Continuing our `Patchy` experiments, in this tutorial we ditch the red-box selector on the timeline and replace it with a gradient. We also try creating our own button background and using that in `StatusEntryView`.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `30-Animations` edition of `Patchy` to use as a starting point.

Step #1: Unwind Red-Box Selector

First, we need to get rid of the red-box selector that we added in an earlier tutorial. As usual, destroying things is easier than creating them.

First, get rid of all `android:background` elements from `Patchy/res/layout/status_entry.xml`, so everything has a transparent background again.

Then, remove the `selectionListener` we added to `Patchy` and any code that references it (e.g., statement in `onCreate()`).

Next, remove the `allWhite` static data member and the references to it from the `TimelineEntryWrapper` constructor.

At this point, we have removed the red-box selector. However, we also turned off the normal Android selector – we will fix that in the next section.

Step #2: Design and Apply a Gradient Selector

Now, we need to create a gradient and use that in our `timeline` `ListView` as the selector.

First, create `Patchy/res/drawable/selector.xml` with the following content:

```
<shape xmlns:android="http://schemas.android.com/apk/res/android"
android:shape="rectangle">
  <gradient
    android:startColor="#4400FF00"
    android:endColor="#FF00FF00"
    android:angle="270"
  />
  <padding
    android:top="2px"
    android:bottom="2px"
  />
  <corners android:radius="10px" />
</shape>
```

Here, we have a rectangle with a green gradient, rounded corners, and a bit of padding.

Now, set the `timeline` `ListView` `android:listSelector` to be `@drawable/selector`.

At this point, if you recompile and reinstall `Patchy`, and use the D-pad to navigate down to the list, you will see the green gradient selector:

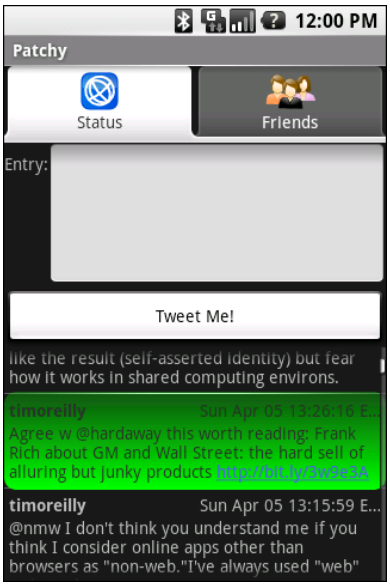


Figure 51. Patchy with a green timeline entry selector

Step #3: Choose a Button Background

If we want to apply a custom button background to one of our buttons, we need to find a PNG background to start with. Perhaps you have a button background you used on a previous (even non-Android) project, or perhaps you are graphically gifted and can draw one yourself. Or, maybe you want to surf OpenClipArt.org or another free clipart site to find a suitable background. Or, download `button_orig.png` from the source code for this project and use the one demonstrated in this tutorial.

Step #4: Make the Button Background be a Nine-Patch

Next, you need to ensure there is a 1-pixel-wide white boundary around the entire image. Your image editing tools may be able to help you do this – if so, save the result as `button.9.png`. If not, download `button.9.png` from the source code for this project and use it.

Step #5: Edit the Nine-Patch Zones

Then, start up the draw9patch program from your SDK installation's `tools/` directory, and browse to and open `button.9.png`. Add black control pixels on the top and left corresponding with the "flat" sides of the button:

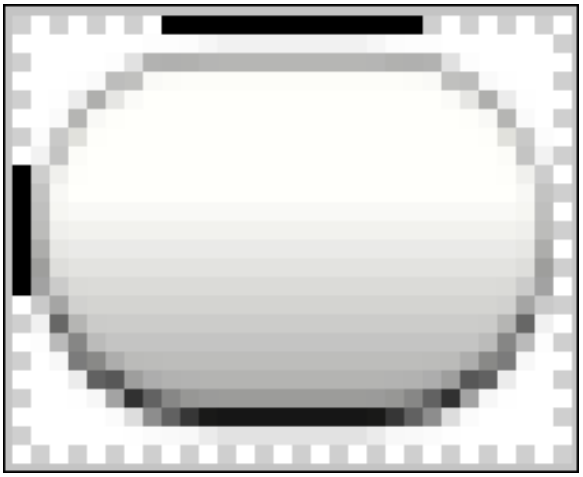


Figure 52. The black bars indicating "stretch zones" in our nine-patch image

Then, save your results.

Step #6: Apply the Button Background

Finally, edit `Patchy/res/layout/status_entry.xml` and add `android:background = "@drawable/button"` to the `send Button` element.

Rebuild and reinstall `Patchy`, and you will see your nine-patch image in action. Note that if you do not have proper transparency set on the PNG, you may wind up with "extra" bits of the image showing through, such as the white space beyond corners in the screenshot below:

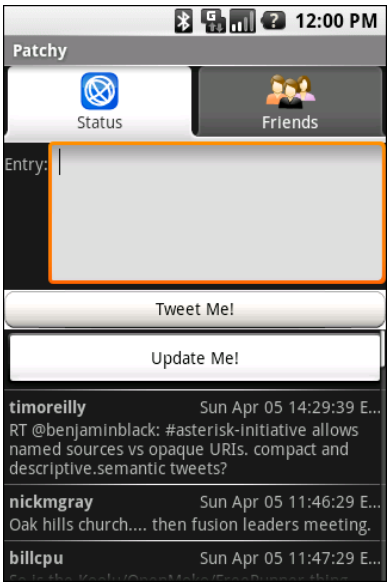


Figure 53. Patchy with a green timeline entry selector

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Do a better job than the author did on having proper PNG transparency set, so the space beyond the "core" of the nine-patch image (e.g., whitespace beyond rounded corners) is transparent.
- Try applying background images to other widgets (e.g., `EditText` for the status entry widget) and see what happens.

Put On Your Happy Face

So far, we have limited ourselves to the stock Droid Sans font for everything. To spice `Patchy` up a bit, we will choose another TrueType font and use it for the `EditText` in the `StatusEntryView`.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `31-Drawable` edition of `Patchy` to use as a starting point.

Step #1: Choose and Install a Font

You will need a TrueType font. Please choose one that is legal to use in this context – not all fonts can be legally copied around. Also, choose something that is decidedly different than Droid Sans. For the samples shown in this tutorial, we are using the Liberation fonts, specifically `LiberationSerif-Regular`.

Once you have the `.TTF` file for your font, put it in the `Patchy/assets/` directory.

Step #2: Use the Font

To use this font, we need to add two lines to the end of our `onFinishInflate()` method in `StatusEntryView`:

```
Typeface face=Typeface.createFromAsset(getContext().getAssets(),
                                       "LiberationSerif-Regular.ttf");
status.setTypeface(face);
```

With this change in place, recompile and reinstall `Patchy`. Try typing in the status update field, and you should see your new font in place:

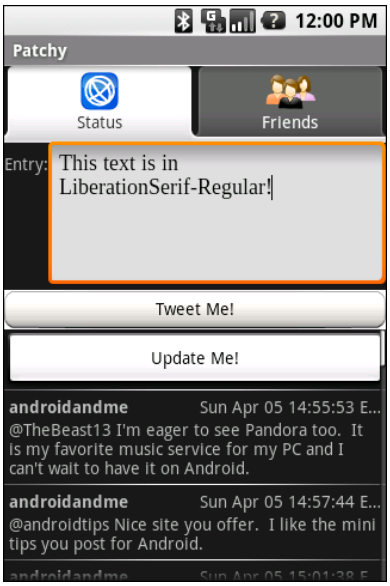


Figure 54. The use of an alternate font in Patchy

If the font did not seem to change, then Android probably had a problem with the font. Some TrueType fonts and Android do not work together well. Try another font and see if that one works. Otherwise, choose the `LiberationSerif-Regular` font shown above, since that is known to work.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Embed a font family and allow the user to choose a font via preferences.
- Apply new fonts to other elements, such as the `Button` widgets.

Photographic Memory

It is time to take a break from `Patchy` and return to our `LunchList` application. One logical thing to add to the restaurant information would be photos: the exterior of the restaurant, favorite dishes, wait staff to avoid, etc. This tutorial will bring us partway there, by allowing users to take a picture while in `LunchList`.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `16-CallSearch` edition of `LunchList` to use as a starting point.

Step #1: Create the Photographer Layout

We need a layout with a full-screen `SurfaceView` to use as the space to display the "preview" – what the camera currently sees.

Create `LunchList/res/layout/photographer.xml` with the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<android.view.SurfaceView
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/preview"
    android:layout_width="fill_parent"
```

```
    android:layout_height="fill_parent"
    >
</android.view.SurfaceView>
```

Step #2: Create the Photographer Class

Next, we need a class that will both connect the live preview to the `SurfaceView`, plus intercept the camera button for use in taking an actual picture. To keep things simple, we will hold off for now on doing anything "for real" with the actual picture.

With that in mind, add a `Photographer` class that looks like the following:

```
package apt.tutorial;

import android.app.Activity;
import android.graphics.PixelFormat;
import android.hardware.Camera;
import android.os.Bundle;
import android.os.Environment;
import android.util.Log;
import android.view.KeyEvent;
import android.view.SurfaceHolder;
import android.view.SurfaceView;

public class Photographer extends Activity {
  private SurfaceView preview=null;
  private SurfaceHolder previewHolder=null;
  private Camera camera=null;

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

    preview=(SurfaceView)findViewById(R.id.preview);
    previewHolder=preview.getHolder();
    previewHolder.addCallback(surfaceCallback);
    previewHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
  }

  @Override
  public boolean onKeyDown(int keyCode, KeyEvent event) {
    if (keyCode==KeyEvent.KEYCODE_CAMERA) {
      takePicture();

      return(true);
    }
```

```
    return(super.onKeyDown(keyCode, event));
  }

  private void takePicture() {
    camera.stopPreview();
    camera.takePicture(null, null, photoCallback);
  }

  SurfaceHolder.Callback surfaceCallback=new SurfaceHolder.Callback() {
    public void surfaceCreated(SurfaceHolder holder) {
      camera=Camera.open();
      camera.setPreviewDisplay(previewHolder);
    }

    public void surfaceChanged(SurfaceHolder holder,
                               int format, int width,
                               int height) {
      Camera.Parameters parameters=camera.getParameters();

      parameters.setPreviewSize(width, height);
      parameters.setPictureFormat(PixelFormat.JPEG);
      camera.setParameters(parameters);
      camera.startPreview();
    }

    public void surfaceDestroyed(SurfaceHolder holder) {
      camera.stopPreview();
      camera.release();
      camera=null;
    }
  };

  Camera.PictureCallback photoCallback=new Camera.PictureCallback() {
    public void onPictureTaken(byte[] data, Camera camera) {

      // do something with the photo JPEG (data[]) here!

      camera.startPreview();
    }
  };
}
```

Here, we initialize our `SurfaceView` in `onCreate()`. When the `SurfaceView` has been created, we connect the camera to it. When the `SurfaceView` size has been set (in `surfaceChanged()`), we configure the `Camera` to work with the size of the `SurfaceView`. When the camera button is clicked, we disable the preview and take a picture, re-enabling the preview in the `PhotoCallback`.

To make this activity work, we need to modify `LunchList/AndroidManifest.xml` to add the `CAMERA` permission and configure the `Photographer` activity:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="apt.tutorial"
      android:versionCode="1"
      android:versionName="1.0">
    <uses-permission android:name="android.permission.CALL_PHONE"/>
    <uses-permission android:name="android.permission.CAMERA" />
    <application android:label="@string/app_name">
        <activity android:name=".LunchList"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
            <intent-filter>
              <action android:name="android.intent.action.SEARCH" />
              <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
            <meta-data android:name="android.app.searchable"
                  android:resource="@xml/searchable" />
            <meta-data android:name="android.app.default_searchable"
                android:value=".LunchList" />
        </activity>
        <activity android:name=".DetailForm">
        </activity>
        <activity android:name=".EditPreferences">
        </activity>
        <activity android:name=".Photographer"
                  android:configChanges="keyboardHidden|orientation"
                  android:screenOrientation="landscape"
                  android:theme="@android:style/Theme.NoTitleBar.Fullscreen">
        </activity>
    </application>
</manifest>
```

Note that we set `Photographer` to be always landscape and use a theme that makes it full-screen, removing the title bar and the status bar.

Step #3: Tie In the Photographer Class

Finally, we need to allow the user to take a picture. Since we (theoretically) want our photos taken with `Photographer` to be associated with the

`Restaurant`, we can add an option menu choice to our `DetailForm`. Here is the revised `LunchList/res/menu/option_detail.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/call"
    android:title="Call"
    android:icon="@android:drawable/ic_menu_call"
  />
  <item android:id="@+id/photo"
    android:title="Take a Photo"
    android:icon="@android:drawable/ic_menu_camera"
  />
</menu>
```

We also need to update `onOptionsItemSelected()` in `DetailForm` to watch for this new menu choice, so modify yours to look like:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.call) {
    if (current.getPhoneNumber()!=null &&
        current.getPhoneNumber().length()>0) {
      String toDial="tel:"+current.getPhoneNumber();

      startActivity(new Intent(Intent.ACTION_CALL,
                               Uri.parse(toDial)));
    }

    return(true);
  }
  else if (item.getItemId()==R.id.photo) {
    startActivity(new Intent(this, Photographer.class));

    return(true);
  }

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

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Store the photos on the SD card. Remember: the emulator does not emulate an SD card by default, so you will need to create an SD card image and tell the emulator to mount it.

- Store the photos on the SD card in a background thread, so the user regains control more quickly.
- Associate photos with the current `Restaurant` and be able to view them later (perhaps via a `Gallery`).

Sensing a Disturbance

Continuing our additions to `LunchList`, we really need a solution to the age-old problem of deciding where to go for lunch. For that, a good old-fashioned random selection would be a fine starting point. But, to make it more entertaining, we should trigger the selection by shaking the device. Shake the phone when on the list of restaurants, and a randomly-selected `Restaurant` will be shown via its `DetailForm`.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `33-Camera` edition of `LunchList` to use as a starting point.

Step #1: Implement a Shaker

We need something that hooks into the `SensorManager` and watches for shaking events, where a "shake" is defined as a certain percentage over Earth's gravity, as determined by calculating the total force via the square root of the sum of the squares of all three dimensional forces.

And if that was total gibberish to you, this is why humankind has developed encapsulation and copy-and-paste.

Create `LunchList/src/apt/tutorial/Shaker.java` with the following source:

```
package apt.tutorial;

import android.content.Context;
import android.hardware.SensorListener;
import android.hardware.SensorManager;
import android.os.SystemClock;

public class Shaker {
  private SensorManager sensor=null;
  private long lastShakeTimestamp=0;
  private double threshold=1.0d;
  private long gap=0;
  private Shaker.Callback cb=null;

  public Shaker(Context ctxt, double threshold, long gap,
                Shaker.Callback cb) {
    this.threshold=threshold;
    this.gap=gap;
    this.cb=cb;

    sensor=(SensorManager)ctxt.getSystemService(Context.SENSOR_SERVICE);
    sensor.registerListener(listener,
                            SensorManager.SENSOR_ACCELEROMETER);
  }

  public void close() {
    sensor.unregisterListener(listener);
  }

  private void isShaking() {
    long now=SystemClock.uptimeMillis();

    if (lastShakeTimestamp==0) {
      lastShakeTimestamp=now;

      if (cb!=null) {
        cb.shakingStarted();
      }
    }
    else {
      lastShakeTimestamp=now;
    }
  }

  private void isNotShaking() {
    long now=SystemClock.uptimeMillis();

    if (lastShakeTimestamp>0) {
      if (now-lastShakeTimestamp>gap) {
        lastShakeTimestamp=0;

        if (cb!=null) {
```

```
          cb.shakingStopped();
        }
      }
    }
  }

  public interface Callback {
    void shakingStarted();
    void shakingStopped();
  }

  private SensorListener listener=new SensorListener() {
    public void onSensorChanged(int sensor, float[] values) {
      if (sensor==SensorManager.SENSOR_ACCELEROMETER) {
        double netForce=Math.pow(values[SensorManager.DATA_X], 2.0);

        netForce+=Math.pow(values[SensorManager.DATA_Y], 2.0);
        netForce+=Math.pow(values[SensorManager.DATA_Z], 2.0);

        if (threshold<(Math.sqrt(netForce)/SensorManager.GRAVITY_EARTH)) {
          isShaking();
        }
        else {
          isNotShaking();
        }
      }
    }

    public void onAccuracyChanged(int sensor, int accuracy) {
      // unused
    }
  };
}
```

The `Shaker` class takes four parameters: a `Context` (used to get the `SensorManager`), the percentage of Earth's gravity that is considered a "shake", how long the acceleration is below that level before a shaking event is considered over, and a callback object to alert somebody about shaking starting and stopping. The math to figure out if, for a given amount of acceleration, we are shaking, is found in the `SensorListener` callback object.

Step #2: Hook Into the Shaker

Given that you magically now have a `Shaker` object, we need to tie it into the `LunchList` activity.

First, implement a `Shaker.Callback` object named `onShake` in `LunchList`:

```
private Shaker.Callback onShake=new Shaker.Callback() {
  public void shakingStarted() {
  }

  public void shakingStopped() {
  }
};
```

Right now, we do not do anything with the shake events, though we will address that shortcoming in the next section.

Then, add a `Shaker` data member to `LunchList` and initialize it somewhere in `onCreate()` by adding this line:

```
shaker=new Shaker(this, 1.15d, 500, onShake);
```

We also need to release our `Shaker` in `onDestroy()`:

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

  model.close();
  db.close();
  shaker.close();
}
```

Now when you shake the device, it will invoke our do-nothing callback.

Step #3: Make a Random Selection on a Shake

To actually choose a `Restaurant`, we need to ask our `RestaurantAdapter` how many `Restaurant` objects there are, then use `Math.random()` to pick one. We can then package that in an `Intent` and start our `DetailForm` on that `Restaurant`.

With that in mind, here is a revised implementation of the `onShake` callback object:

```
private Shaker.Callback onShake=new Shaker.Callback() {
  public void shakingStarted() {
```

```
    int selection=(int)(adapter.getCount()*Math.random());

    Intent i=new Intent(LunchList.this, DetailForm.class);

    i.putExtra(ID_EXTRA,
               String.valueOf(adapter.getItemId(selection)));
    startActivity(i);
  }

  public void shakingStopped() {
  }
};
```

If you rebuild and reinstall the application on a device, you can randomly choose a `Restaurant` by giving the phone a solid shake.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Store the GPS coordinates of the `Restaurant` via the `DetailForm`. Then, given your current position and the location of the Restaurant, present a compass to help point you in the right direction, in case you forget where the restaurant is. WARNING: this may involve icky math.
- Attempt to use the accelerometer to measure your current speed while walking or jogging. Remind yourself partway through why, exactly, you elected not to be a physics major in college. If you were a physics major in college, the author offers his sincere condolences.

TUTORIAL 35

Messages From The Great Beyond

In this tutorial, we will replace the callback system we used in previous editions of `Patchy` with a broadcast `Intent`. This allows `TwitterMonitor` to alert applications about new status updates without there having to be an explicit connection between them.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `32-Fonts` edition of `Patchy` to use as a starting point.

Step #1: Broadcast the Intent

Sending a broadcast `Intent` is rather simple, particularly for an application like `TwitterMonitor` that already collects the information to broadcast (in this case, friend, status, and creation date).

First, we need to choose a name for the action of the broadcast `Intent`. We need something that will not collide with other such names, so it is best to use something that has our namespace in it. Similarly, we need to have

names for any values we are going to store as "extras" in our Intent, and ideally those are namespaced as well.

With that in mind, add the following data members to `TwitterMonitor`:

```
public static final String STATUS_UPDATE="apt.tutorial.three.STATUS_UPDATE";
public static final String FRIEND="apt.tutorial.three.FRIEND";
public static final String STATUS="apt.tutorial.three.STATUS";
public static final String CREATED_AT="apt.tutorial.three.CREATED_AT";
```

Notice that we declare them as `public static final`. In principle, if we had a receiver of these broadcasts that was compiled in our project, the receiver could reference those values directly. As we will see in the next section, though, that does not work out if the receiver is in some other project.

Then, we need to actually broadcast the `Intent`, instead of calling the callback. To do that, replace your existing `poll()` implementation with the following:

```
synchronized private void poll(Account l) {
  try {
    Twitter client=new Twitter(l.user, l.password);
    List<Twitter.Status> timeline=client.getFriendsTimeline();

    for (Twitter.Status s : timeline) {
      if (!seenStatus.contains(s.id)) {
        try {
          Intent broadcast=new Intent(STATUS_UPDATE);

          broadcast.putExtra(FRIEND, s.user.screenName);
          broadcast.putExtra(STATUS, s.text);
          broadcast.putExtra(CREATED_AT,
                             s.createdAt.toString());

          sendBroadcast(broadcast);

          seenStatus.add(s.id);
        }
        catch (Throwable t) {
          Log.e("TwitterMonitor", "Exception in callback", t);
        }

        if (bff.contains(s.user.screenName)) {
          notify(s.user.screenName);
        }
      }
    }
```

```
  }
  catch (Throwable t) {
    Log.e("TwitterMonitor", "Exception in poll()", t);
  }
}
```

We create a fresh `Intent` on our defined action, populate our variable data as "extras", and call `sendBroadcast()` to send it off.

Step #2: Catch and Use the Intent

Receiving a broadcast `Intent` is similarly simple.

First, we need to decide when we want our receiver to be active and alive. In some situations, where the broadcasts are merely advisory, we can enable them in `onResume()` and disable them in `onPause()`, so we consume no CPU time processing broadcasts when our activity is not visible. In this case, so we do not miss an all-important tweet, we should ensure the receiver is active whenever our account registered with `TwitterMonitor` is active.

Since `Patchy` cannot see the constants defined for the Intent action and "extras" in `TwitterMonitor`, we need to clone those lines in `Patchy` itself:

```
public static final String STATUS_UPDATE="apt.tutorial.three.STATUS_UPDATE";
public static final String FRIEND="apt.tutorial.three.FRIEND";
public static final String STATUS="apt.tutorial.three.STATUS";
public static final String CREATED_AT="apt.tutorial.three.CREATED_AT";
```

Then, we need to register a receiver in `onCreate()` before we bind to our service:

```
registerReceiver(receiver, new IntentFilter(STATUS_UPDATE));
```

And, we need to unregister said receiver after we unbind from the service:

```
unregisterReceiver(receiver);
```

Of course, none of this will work without a receiver itself. We want the receiver to do what our callback object does, except that instead of getting

the data as method parameters, we get it as "extras" on the broadcast `Intent`. Here is one implementation of receiver that will accomplish this end:

```
private BroadcastReceiver receiver=new BroadcastReceiver() {
  public void onReceive(Context context,
                        final Intent intent) {
    runOnUiThread(new Runnable() {
      public void run() {
        String friend=intent.getStringExtra(FRIEND);
        String createdAt=intent.getStringExtra(CREATED_AT);
        String status=intent.getStringExtra(STATUS);

        adapter.insert(new TimelineEntry(friend,
                                         createdAt,
                                         status), 0);
      }
    });
  }
};
```

If we intend to keep support for both the callback and the broadcast `Intent`, we might consider refactoring our code, such that the `TimelineEntry` addition is centrally defined.

At this point, recompile and reinstall both `TMonitor` and `Patchy`. `Patchy` should run as it did before, just via broadcast `Intent` objects instead of the callback.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Get rid of the callback API. Since the service tracks accounts by the callback object (it is the key to the `Map`), you will need to restructure how the service tracks those accounts, such as using the screen name. When you change the AIDL, be sure to do so in both the activity and the service!
- Create a separate "sniffer" project that listens for the same broadcast `Intent`. Confirm that it too receives the broadcasts from `TwitterMonitor`.

A Time For Quiet Introspection

Android offers a nice set of facilities for you to figure out what you can do with a given Uri. If the Uri represents a collection, you can allow the user to pick a specific piece of content out of the collection. And you can inject menu choices for any sort of action on the Uri as part of your option menu. In this tutorial, we will look at both of those capabilities.

Step-By-Step Instructions

This tutorial starts a new application, independent from the `LunchList` and `Patchy` applications developed in the preceding tutorials. Hence, we will have you create a new application from scratch.

Step #1: Create a Stub Project

Using Eclipse or `activitycreator`, make a project named `IntentExplorer` with a stub activity named `apt.tutorial.four.IntentExplorer`. The generated activity class should resemble the following:

```
package apt.tutorial.four;

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

public class IntentExplorer extends Activity
{
 /** Called when the activity is first created. */
```

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

Step #2: Create a Layout

The goal of `IntentExplorer` is to allow you to choose a content provider, pick a piece of content, then take actions on that piece of content via the options menu. That means we need a layout that lets us accomplish those ends.

With that in mind, create `IntentExplorer/res/layout/main.xml` with the following content:

```
<?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://com.google.provider.NotePad/notes"
  />
  <Button
    android:id="@+id/pick"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Pick a Piece of Content"
  />
  <TextView android:id="@+id/uri"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:singleLine="false"
  />
</LinearLayout>
```

We have an `EditText` for the user to fill in the base `Uri` for a content provider. We also have a "Pick" button to let the user pick a piece of content from that provider, and a `TextView` to show the `Uri` of the chosen content.

Step #3: Pick Content on Button Click

Next, we need to arrange to get control on the button click and let the user pick a piece of content from the collection entered into the `EditText` widget. To pick a piece of content, we need to start an activity for an `ACTION_PICK` `Intent`, with the `Intent` `Uri` set to be the collection we want. More importantly, we need to use `startActivityForResult()`, so we can get the chosen piece of content returned to us.

Modify the `IntentExplorer` activity to look like the following:

```
package apt.tutorial.four;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.view.Menu;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

public class IntentExplorer extends Activity {
  static final int PICK_REQUEST=1337;
  private EditText type=null;
  private TextView chosenContent=null;
  private Uri chosenUri=null;

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

    type=(EditText)findViewById(R.id.type);
    chosenContent=(TextView)findViewById(R.id.uri);

    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) {
        chosenUri=data.getData();
        chosenContent.setText(chosenUri.toString());
      }
    }
  }
}
```

When we get control in `onActivityResult()`, we hold onto the chosen `Uri` (for later use) and fill in its string representation into the `TextView` to display to the user.

Step #4: Customize the Option Menu

Once we have a `Uri`, we can use it to add options to our options menu, based on what is all available for that piece of content.

Add the following implementation of `onPrepareOptionsMenu()` to `IntentExplorer`:

```
@Override
public boolean onPrepareOptionsMenu(Menu menu) {
  Intent i=new Intent(null, chosenUri);

  i.addCategory(Intent.CATEGORY_ALTERNATIVE);

  int c=menu.addIntentOptions(Menu.CATEGORY_ALTERNATIVE, 0, 0,
                      null, null, i, 0, null);

  return(super.onPrepareOptionsMenu(menu));
}
```

Here, we create a no-action `Intent` with our `Uri`, but set the `Intent` category to `CATEGORY_ALTERNATIVE`, indicating we want activities published as being designed to be injected into other applications, like we are doing here. Then, we call `addIntentOptions()` on the supplied `Menu` object, specifying a bunch of default values plus our custom-crafted `Intent`.

At this point, you can recompile and reinstall the application...though you may need to do a bit more work to actually use it.

Step #5: Trying it Out

Picking a piece of content with `IntentExplorer` is easy. You could use `content://contacts/people` to pick a contact from the Contacts application, for example.

The problem is finding content that offers up `CATEGORY_ALTERNATIVE` menu choices. Most of the built-in applications lack this facility.

One easy way to get an application onto your device or emulator that does support it is to compile and install the Notepad sample application that accompanied your SDK. The default value we put in our layout is the base content `Uri` for Notepad.

If you go this route, add a few notes, then run `IntentExplorer` to bring up the initial UI:

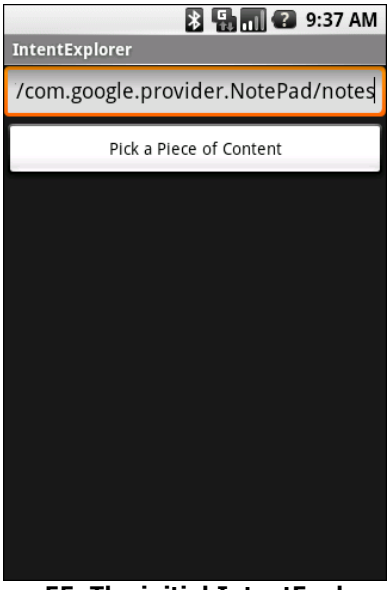


Figure 55. The initial IntentExplorer UI

Then, click Pick and choose a note:

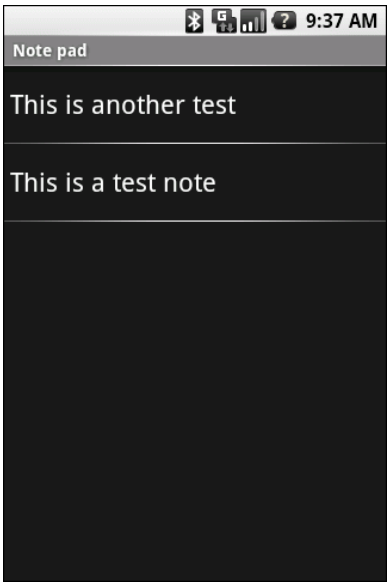


Figure 56. Notepad's implementation of ACTION_PICK

Due to flaws in the `ACTION_PICK` implementation of Notepad, after clicking on a note, you need to press the back button to record the selection.

Back in `IntentExplorer`, you will see the `Uri` of your chosen note below the Pick button:

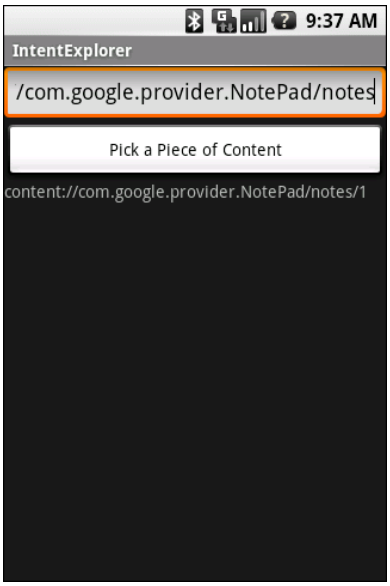


Figure 57. IntentExplorer after choosing a piece of content

Clicking the [MENU] button will display the vast array of available options for you to take on this piece of content:

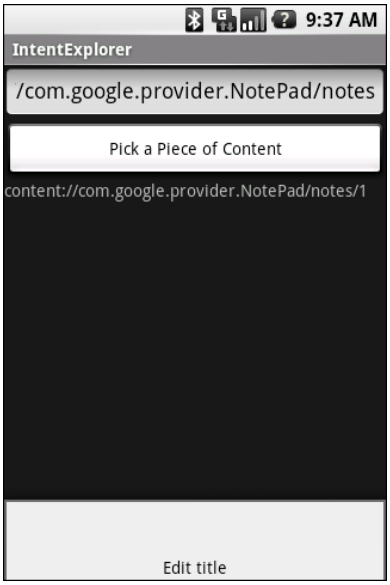


Figure 58. IntentExplorer's option menu, showing an option supplied by Notepad

Choosing Edit Title brings up a dialog to let you edit the note's title:

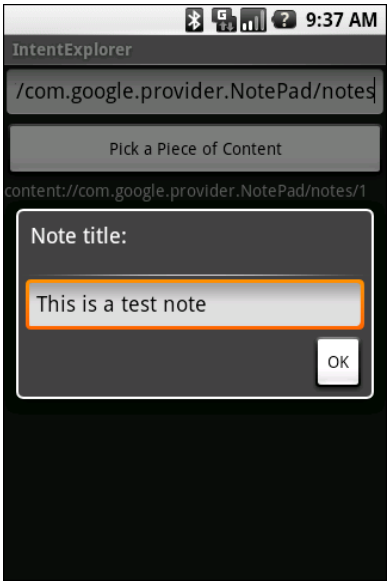


Figure 59. Notepad's edit-title dialog, shown over the IntentExplorer UI

You did not implement that dialog – it is supplied by the Notepad application. All you did was arrange for Notepad to have menu choices in `IntentExplorer`.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Find other applications that offer content and `CATEGORY_ALTERNATIVE` actions.
- After you add the content provider to `LunchList` in a later tutorial, also add support for `ACTION_PICK` (so `LunchList` will return a selected `Restaurant` rather than open up a `DetailForm` on it). Then, arrange for the `DetailForm` activity to support `CATEGORY_ALTERNATIVE` on the `Restaurant` MIME type, and try using `IntentExplorer` to pick a `Restaurant` and view its details.

Contacting Twitter

In this tutorial, we tie Android contacts to `Patchy`, storing Twitter screen names in the contacts engine and highlighting those Twitter status updates that come from our contacts.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `35-IntentFilters` edition of `Patchy` to use as a starting point.

Step #1: Fake the Contact Data

Android's contact database does not have a built-in spot to record the Twitter screen name of the contact.

Hopefully, you have gotten over the horror of that news. If so, the hack we will use for this tutorial is to create a fictitious organization type called "Twitter", and use the organization name as the spot for the Twitter screen name.

To do this, pick a contact, or create a new one if you do not have any. Fill in the person's display name, then scroll down and click the large "More Info" button:

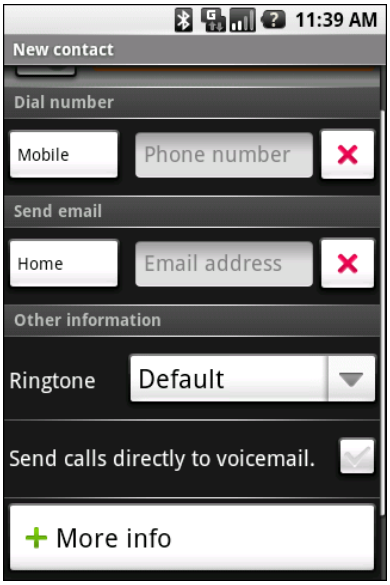


Figure 60. The contact detail form, showing the "More Info" button

Figure 61. The selection dialog that appears after clicking "More Info"

Next, expand the Other category to uncover the Organization option:

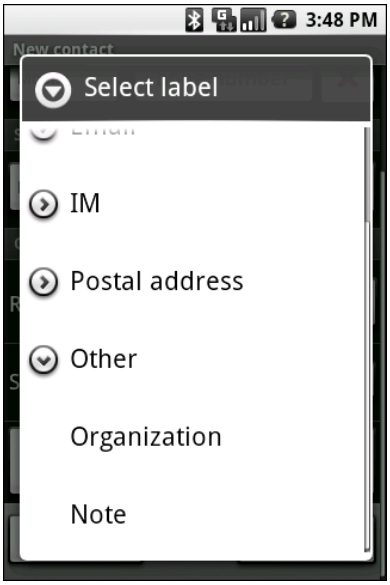


Figure 62. The selection dialog, showing the Organization option

Click on Organization, which will add a blank organization set of widgets to your contact detail form:

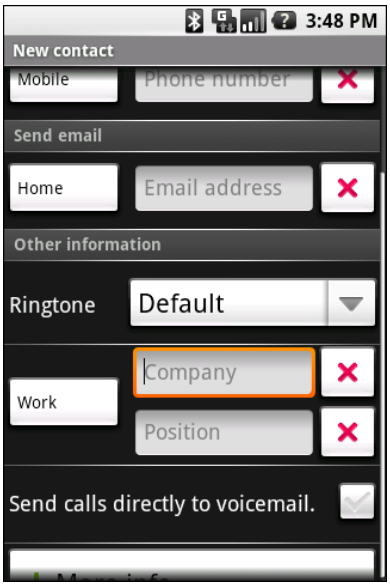


Figure 63. The contact detail form, showing the blank organization

Click the Work button to bring up the roster of available organization types:

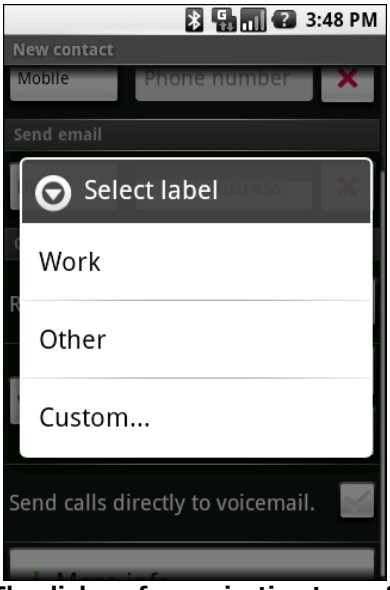


Figure 64. The dialog of organization types for contacts

Choose "Custom...", then fill in "Twitter" (sans quotes) as the organization type:

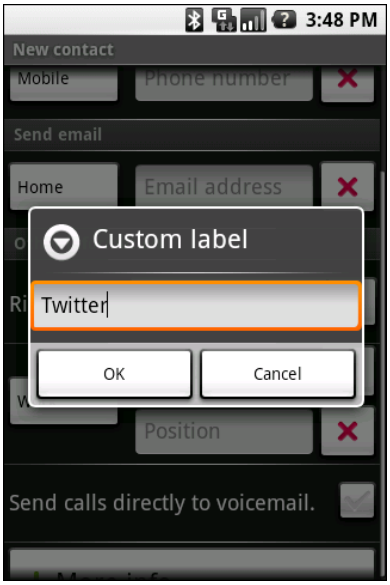


Figure 65. The organization type dialog, with "Twitter" entered

Click OK, then fill in the Twitter screen name for this contact in the organization field:

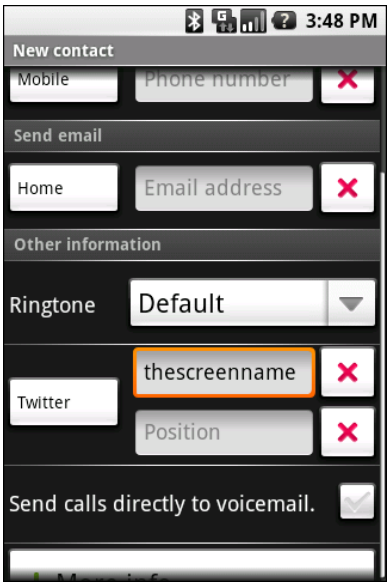


Figure 66. The contact detail form, showing the "Twitter" organization type and "thescreename" as the Twitter screen name

Choose Save from the option menu, and you are set. You will need one or more contacts set up with Twitter screen names. In particular, you need contacts that will have status updates in your timeline.

Step #2: Design the Highlight

Next, we need to decide how we wish to highlight those friends who are in our contacts database. A simple highlight that we can apply is to put a background color on the friend's name – that will help friends who are contacts to stand out from the rest.

To do that, we first need to amend `TimelineEntry`, so it knows whether or not it is a contact – add the following data member to the `TimelineEntry` inner class of `Patchy`:

```
boolean isContact=false;
```

Then, we need to modify `TimelineEntryWrapper` so it looks at the `isContact` property of `TimelineEntry` and sets the background accordingly, so change `populateFrom()` in `TimelineEntryWrapper` to be:

```
void populateFrom(TimelineEntry s) {
  getFriend().setText(s.friend);
  getCreatedAt().setText(s.createdAt);
  getStatus().setText(s.status);

  if (s.isContact) {
    getFriend().setBackgroundColor(0xFF0000FF);
  }
  else {
    getFriend().setBackgroundColor(0x00000000);
  }
}
```

You might think that we only need to set the background when the friend is a contact. However, since rows get recycled, if we do not reset the background when the friend is not a contact, soon all our rows will be highlighted, mostly in error.

Step #3: Find and Highlight Matching Contacts

Of course, our `isContact` property in `TimelineEntry` is always set to be `false`, which is not terribly helpful. Instead, we need to do a lookup to see if a friend is actually a contact, so we can set that flag appropriately. We do not need any data about the contact – all we need to know is if there exists a contact with the appropriate Twitter "organization" and screen name.

First, add the `READ_CONTACTS` permission to the `AndroidManifest.xml` file for `Patchy`. Without this, we cannot find out if a friend is a contact.

Next, add a `PROJECTION` to `Patchy`:

```
private static final String[] PROJECTION=new String[] {
                                    Contacts.People._ID,
                                                };
```

This just says that the only column we want back is the contact's ID.

Finally, modify the `TimelineEntry` constructor to be as follows:

```
TimelineEntry(String friend, String createdAt,
              String status) {
  this.friend=friend;
  this.createdAt=createdAt;
  this.status=status;

  String[] args={friend};
  StringBuffer query=new StringBuffer(Contacts.OrganizationColumns.LABEL);

  query.append("='Twitter' AND ");
  query.append(Contacts.OrganizationColumns.COMPANY);
  query.append("=?");

  Cursor c=managedQuery(Contacts.Organizations.CONTENT_URI,
                        PROJECTION,
                        query.toString(),
                        args,
                        Contacts.People.DEFAULT_SORT_ORDER);

  if (c.getCount()>0) {
    this.isContact=true;
  }

  c.close();
}
```

We use `managedQuery()` to get a `Cursor` representing our contact, if any. All we do is look to see if we got one (or, conceivably, more) matches on our screen name, and use that information to set the value of `isContact`.

The net result is that any friends whose Twitter screen names are in our contacts will show up with their screen names on a blue background.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Provide a means from the `Patchy` UI to see the contact information for a contact from whom we have received a status update.
- Cache the matching contacts for a short period of time, to reduce contact lookups when new status updates come in.

- Provide a means from the `Patchy` UI to "re-tweet" both inside Twitter and by forwarding the tweet (and its URL) to contacts via email or SMS.

Makin' Content

In this tutorial, we will make the `LunchList` roster of `Restaurant` objects be available via a content provider, to help expand the ecosystem of possible restaurant list management tools...or, possibly, just as a proof of concept.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `34-Sensors` edition of `LunchList` to use as a starting point.

Step #1: Define the Restaurant Provider

We will build up a `RestaurantProvider` implementation in the following subsections:

Define the Class

Create `LunchList/src/apt/tutorial/RestaurantProvider.java` with the following declaration:

```
public class RestaurantProvider extends ContentProvider {
```

Implement onCreate()

Like an activity or service, content providers have an `onCreate()` entry point – implement one on `RestaurantProvider` as follows:

```
@Override
public boolean onCreate() {
  db=(new RestaurantSQLiteHelper(getContext())).getWritableDatabase();

  return((db == null) ? false : true);
}
```

This implementation assumes the existence of a `SQLiteDatabase` data member named `db`, which we initialize to have access to our application database via `RestaurantSQLiteHelper`.

Implement getType()

The `getType()` method needs to return the MIME type for a given `Uri`, in particular distinguishing between collections and individual instances. Implement `getType()` on `RestaurantProvider` as follows:

```
@Override
public String getType(Uri url) {
  if (isCollectionUri(url)) {
    return(getCollectionType());
  }

  return(getSingleType());
}
```

This delegates the smarts to three private methods. The `getCollectionType()` and `getSingleType()` methods are easy, so define them as follows:

```
private String getCollectionType() {
  return("vnd.android.cursor.dir/vnd.apt.tutorial.restaurant");
}

private String getSingleType() {
  return("vnd.android.cursor.item/vnd.apt.tutorial.restaurant");
}
```

The left portion of the MIME type is dictated by Android; the right portion is something unique we use in our application.

We also need `isCollectionUri()`:

```
private boolean isCollectionUri(Uri url) {
  return(MATCHER.match(url)==RESTAURANTS);
}
```

This takes advantage of a `UriMatcher` static instance:

```
private static final int RESTAURANTS=1;
private static final int RESTAURANT_ID=2;
private static final UriMatcher MATCHER;
```

which needs to be initialized:

```
static {
  MATCHER=new UriMatcher(UriMatcher.NO_MATCH);
  MATCHER.addURI("apt.tutorial.RestaurantProvider",
                 "restaurants", RESTAURANTS);
  MATCHER.addURI("apt.tutorial.RestaurantProvider",
                 "restaurants/#", RESTAURANT_ID);
}
```

Implement query()

Next, we need to implement the `query()` method, to handle queries for the content provider:

```
@Override
public Cursor query(Uri url, String[] projection, String selection,
                    String[] selectionArgs, String sort) {
  SQLiteQueryBuilder qb=new SQLiteQueryBuilder();
  String orderBy;

  qb.setTables(getTableName());

  if (isCollectionUri(url)) {
    qb.setProjectionMap(getDefaultProjection());
  }
  else {
    qb.appendWhere(getIdColumnName()+"="+url.getPathSegments().get(1));
  }
```

```
  if (TextUtils.isEmpty(sort)) {
    orderBy=getDefaultSortOrder();
  }
  else if (sort.startsWith("ORDER BY")) {
    orderBy=sort.substring(9);
  }
  else {
    orderBy=sort;
  }

  Cursor c=qb.query(db, projection, selection, selectionArgs,
                    null, null, orderBy);
  c.setNotificationUri(getContext().getContentResolver(), url);

  return(c);
}
```

This method populates a `SQLiteQueryBuilder` and uses it to actually invoke the query itself.

This method relies upon several other private methods, including `getDefaultProjection()`:

```
private HashMap<String, String> getDefaultProjection() {
  return(DEFAULT_PROJECTION);
}
```

which in turn relies upon a static `DEFAULT_PROJECTION`:

```
public static HashMap<String, String> DEFAULT_PROJECTION;
```

which needs to be statically initialized:

```
public static final class Columns implements BaseColumns {
  public static final Uri CONTENT_URI
      =Uri.parse("content://apt.tutorial.RestaurantProvider/restaurants");
  public static final String NAME="name";
  public static final String NOTES="notes";
  public static final String ADDRESS="address";
  public static final String TYPE="type";
  public static final String PHONENUMBER="phoneNumber";
  public static final String DEFAULT_SORT_ORDER=NAME;
}

static {
  DEFAULT_PROJECTION=new HashMap<String, String>();
  DEFAULT_PROJECTION.put(RestaurantProvider.Columns._ID,
```

```
                                  RestaurantProvider.Columns._ID);
  DEFAULT_PROJECTION.put(RestaurantProvider.Columns.NAME,
                                  RestaurantProvider.Columns.NAME);
  DEFAULT_PROJECTION.put(RestaurantProvider.Columns.NOTES,
                                  RestaurantProvider.Columns.NOTES);
  DEFAULT_PROJECTION.put(RestaurantProvider.Columns.ADDRESS,
                                  RestaurantProvider.Columns.ADDRESS);
  DEFAULT_PROJECTION.put(RestaurantProvider.Columns.TYPE,
                                  RestaurantProvider.Columns.TYPE);
  DEFAULT_PROJECTION.put(RestaurantProvider.Columns.PHONENUMBER,
                                  RestaurantProvider.Columns.PHONENUMBER);
}
```

We also need `getIdColumnName()`:

```
private String getIdColumnName() {
  return("_id");
}
```

as well as `getDefaultSortOrder()`:

```
private String getDefaultSortOrder() {
  return(RestaurantProvider.Columns.DEFAULT_SORT_ORDER);
}
```

which in turn relies upon the `DEFAULT_SORT_ORDER` shown above.

Note that with the sort order, we really expect the sort order to be defined as everything after the `ORDER BY` of a regular SQL query. If we are supplied an `ORDER BY` by accident, we strip it off.

Implement insert()

Next up: add the following implementation of `insert()` to `RestaurantProvider`:

```
@Override
public Uri insert(Uri url, ContentValues initialValues) {
  long rowID;
  ContentValues values;

  if (initialValues!=null) {
    values=new ContentValues(initialValues);
  }
```

```
  else {
    values=new ContentValues();
  }

  if (!isCollectionUri(url)) {
    throw new IllegalArgumentException("Unknown URL " + url);
  }

  for (String colName : getRequiredColumns()) {
    if (values.containsKey(colName)==false) {
      throw new IllegalArgumentException("Missing column: "+colName);
    }
  }

  rowID=db.insert(getTableName(), getNullColumnHack(), values);

  if (rowID>0) {
    Uri uri=ContentUris.withAppendedId(getContentUri(), rowID);
    getContext().getContentResolver().notifyChange(uri, null);

    return(uri);
  }

  throw new SQLException("Failed to insert row into " + url);
}
```

It validates the required columns, populates a `ContentValues`, and does the actual insert into the database.

It relies on several other methods, including `getRequiredColumns()`:

```
private String[] getRequiredColumns() {
  return(new String[] {RestaurantProvider.Columns.NAME});
}
```

It also needs `getTableName()`:

```
private String getTableName() {
  return("restaurants");
}
```

and `getNullColumnHack()`:

```
private String getNullColumnHack() {
  return(RestaurantProvider.Columns.NAME);
}
```

to go along with `getContentUri()`:

```
private Uri getContentUri() {
  return(RestaurantProvider.Columns.CONTENT_URI);
}
```

Implement update()

Then, implement `update()` on `RestaurantProvider`:

```
@Override
public int update(Uri url, ContentValues values,
                  String where, String[] whereArgs) {
  int count;

  if (isCollectionUri(url)) {
    count=db.update(getTableName(), values, where, whereArgs);
  }
  else {
    String segment=url.getPathSegments().get(1);
    count=db
        .update(getTableName(), values, getIdColumnName()+"="
            + segment
            + (!TextUtils.isEmpty(where) ? " AND (" + where
                + ')' : ""), whereArgs);
  }

  getContext().getContentResolver().notifyChange(url, null);

  return(count);
}
```

This builds up the various bits of the SQL `UPDATE` statement, in particular handling the complexities of a `WHERE` clause defined in part by a passed-in parameter and, possibly, in part by the row ID on the supplied `Uri`.

Implement delete()

Finally, implement `delete()`, which looks a lot like `update()`:

```
@Override
public int delete(Uri url, String where, String[] whereArgs) {
  int count;
  long rowId=0;
```

```
  if (isCollectionUri(url)) {
    count=db.delete(getTableName(), where, whereArgs);
  }
  else {
    String segment=url.getPathSegments().get(1);
    rowId=Long.parseLong(segment);
    count=db
        .delete(getTableName(), getIdColumnName()+"="
            + segment
            + (!TextUtils.isEmpty(where) ? " AND (" + where
                + ')' : ""), whereArgs);
  }

  getContext().getContentResolver().notifyChange(url, null);

  return(count);
}
```

Here is an entire implementation of `RestaurantProvider`:

```
package apt.tutorial;

import android.content.ContentProvider;
import android.content.ContentUris;
import android.content.ContentValues;
import android.content.UriMatcher;
import android.content.res.Resources;
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteQueryBuilder;
import android.net.Uri;
import android.provider.BaseColumns;
import android.text.TextUtils;
import java.util.HashMap;

public class RestaurantProvider extends ContentProvider {
  public static HashMap<String, String> DEFAULT_PROJECTION;
  private static final int RESTAURANTS=1;
  private static final int RESTAURANT_ID=2;
  private static final UriMatcher MATCHER;

  static {
    MATCHER=new UriMatcher(UriMatcher.NO_MATCH);
    MATCHER.addURI("apt.tutorial.RestaurantProvider",
                   "restaurants", RESTAURANTS);
    MATCHER.addURI("apt.tutorial.RestaurantProvider",
                   "restaurants/#", RESTAURANT_ID);
  }

  public static final class Columns implements BaseColumns {
    public static final Uri CONTENT_URI
        =Uri.parse("content://apt.tutorial.RestaurantProvider/restaurants");
```

```
    public static final String NAME="name";
    public static final String NOTES="notes";
    public static final String ADDRESS="address";
    public static final String TYPE="type";
    public static final String PHONENUMBER="phoneNumber";
    public static final String DEFAULT_SORT_ORDER=NAME;
  }

  static {
    DEFAULT_PROJECTION=new HashMap<String, String>();
    DEFAULT_PROJECTION.put(RestaurantProvider.Columns._ID,
                           RestaurantProvider.Columns._ID);
    DEFAULT_PROJECTION.put(RestaurantProvider.Columns.NAME,
                           RestaurantProvider.Columns.NAME);
    DEFAULT_PROJECTION.put(RestaurantProvider.Columns.NOTES,
                           RestaurantProvider.Columns.NOTES);
    DEFAULT_PROJECTION.put(RestaurantProvider.Columns.ADDRESS,
                           RestaurantProvider.Columns.ADDRESS);
    DEFAULT_PROJECTION.put(RestaurantProvider.Columns.TYPE,
                           RestaurantProvider.Columns.TYPE);
    DEFAULT_PROJECTION.put(RestaurantProvider.Columns.PHONENUMBER,
                           RestaurantProvider.Columns.PHONENUMBER);
  }

  private SQLiteDatabase db;

  @Override
  public boolean onCreate() {
    db=(new RestaurantSQLiteHelper(getContext())).getWritableDatabase();

    return((db == null) ? false : true);
  }

  @Override
  public Cursor query(Uri url, String[] projection, String selection,
                      String[] selectionArgs, String sort) {
    SQLiteQueryBuilder qb=new SQLiteQueryBuilder();
    String orderBy;

    qb.setTables(getTableName());

    if (isCollectionUri(url)) {
      qb.setProjectionMap(getDefaultProjection());
    }
    else {
      qb.appendWhere(getIdColumnName()+"="+url.getPathSegments().get(1));
    }

    if (TextUtils.isEmpty(sort)) {
      orderBy=getDefaultSortOrder();
    }
    else if (sort.startsWith("ORDER BY")) {
      orderBy=sort.substring(9);
    }
```

```
    else {
      orderBy=sort;
    }

    Cursor c=qb.query(db, projection, selection, selectionArgs,
                      null, null, orderBy);
    c.setNotificationUri(getContext().getContentResolver(), url);

    return(c);
  }

  @Override
  public String getType(Uri url) {
    if (isCollectionUri(url)) {
      return(getCollectionType());
    }

    return(getSingleType());
  }

  @Override
  public Uri insert(Uri url, ContentValues initialValues) {
    long rowID;
    ContentValues values;

    if (initialValues!=null) {
      values=new ContentValues(initialValues);
    }
    else {
      values=new ContentValues();
    }

    if (!isCollectionUri(url)) {
      throw new IllegalArgumentException("Unknown URL " + url);
    }

    for (String colName : getRequiredColumns()) {
      if (values.containsKey(colName)==false) {
        throw new IllegalArgumentException("Missing column: "+colName);
      }
    }

    rowID=db.insert(getTableName(), getNullColumnHack(), values);

    if (rowID>0) {
      Uri uri=ContentUris.withAppendedId(getContentUri(), rowID);
      getContext().getContentResolver().notifyChange(uri, null);

      return(uri);
    }

    throw new SQLException("Failed to insert row into " + url);
  }
```

```
@Override
public int delete(Uri url, String where, String[] whereArgs) {
  int count;
  long rowId=0;

  if (isCollectionUri(url)) {
    count=db.delete(getTableName(), where, whereArgs);
  }
  else {
    String segment=url.getPathSegments().get(1);
    rowId=Long.parseLong(segment);
    count=db
        .delete(getTableName(), getIdColumnName()+"="
            + segment
            + (!TextUtils.isEmpty(where) ? " AND (" + where
                + ')' : ""), whereArgs);
  }

  getContext().getContentResolver().notifyChange(url, null);

  return(count);
}

@Override
public int update(Uri url, ContentValues values,
                  String where, String[] whereArgs) {
  int count;

  if (isCollectionUri(url)) {
    count=db.update(getTableName(), values, where, whereArgs);
  }
  else {
    String segment=url.getPathSegments().get(1);
    count=db
        .update(getTableName(), values, getIdColumnName()+"="
            + segment
            + (!TextUtils.isEmpty(where) ? " AND (" + where
                + ')' : ""), whereArgs);
  }

  getContext().getContentResolver().notifyChange(url, null);

  return(count);
}

private boolean isCollectionUri(Uri url) {
  return(MATCHER.match(url)==RESTAURANTS);
}

private HashMap<String, String> getDefaultProjection() {
  return(DEFAULT_PROJECTION);
}

private String getTableName() {
```

```
    return("restaurants");
  }

  private String getIdColumnName() {
    return("_id");
  }

  private String getDefaultSortOrder() {
    return(RestaurantProvider.Columns.DEFAULT_SORT_ORDER);
  }

  private String getCollectionType() {
    return("vnd.android.cursor.dir/vnd.apt.tutorial.restaurant");
  }

  private String getSingleType() {
    return("vnd.android.cursor.item/vnd.apt.tutorial.restaurant");
  }

  private String[] getRequiredColumns() {
    return(new String[] {RestaurantProvider.Columns.NAME});
  }

  private String getNullColumnHack() {
    return(RestaurantProvider.Columns.NAME);
  }

  private Uri getContentUri() {
    return(RestaurantProvider.Columns.CONTENT_URI);
  }
}
```

Step #2: Publish the Provider

Assuming that `RestaurantProvider` compiled cleanly for you, you can now add it to the `AndroidManifest.xml` for the `LunchList` project, so the provider is available for use:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="apt.tutorial"
      android:versionCode="1"
      android:versionName="1.0">
    <uses-permission android:name="android.permission.CALL_PHONE"/>
    <uses-permission android:name="android.permission.CAMERA" />
    <application android:label="@string/app_name">
        <provider android:name=".RestaurantProvider"
            android:authorities="apt.tutorial.RestaurantProvider" />
        <activity android:name=".LunchList"
                  android:label="@string/app_name">
```

```
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
            <intent-filter>
              <action android:name="android.intent.action.SEARCH" />
              <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
            <meta-data android:name="android.app.searchable"
                  android:resource="@xml/searchable" />
            <meta-data android:name="android.app.default_searchable"
                android:value=".LunchList" />
        </activity>
        <activity android:name=".DetailForm">
        </activity>
        <activity android:name=".EditPreferences">
        </activity>
        <activity android:name=".Photographer"
                  android:configChanges="keyboardHidden|orientation"
                  android:screenOrientation="landscape"
                  android:theme="@android:style/Theme.NoTitleBar.Fullscreen">
        </activity>
    </application>
</manifest>
```

Step #3: Use the Provider in the LunchList

Now, we can set about actually using the `RestaurantProvider` in our `LunchList` activity. For now, we will leave the `DetailForm` implementation alone.

First, we really should adjust our preference values, to get rid of the extraneous `ORDER BY` that is no longer needed. Modify `LunchList/res/values/arrays.xml` to remove the `ORDER BY` bits:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string-array name="sort_names">
    <item>By Name, Ascending</item>
    <item>By Name, Descending</item>
    <item>By Type</item>
    <item>By Address, Ascending</item>
    <item>By Address, Descending</item>
  </string-array>
  <string-array name="sort_clauses">
    <item>name ASC</item>
    <item>name DESC</item>
    <item>type, name ASC</item>
    <item>address ASC</item>
    <item>address DESC</item>
```

```
  </string-array>
</resources>
```

Then, we can get rid of our `SQLiteDatabase` instance and all references to it (e.g., the `db.close();` statement in `onDestroy()`), since `LunchList` will be routing all queries through `RestaurantProvider`.

Finally, in `initList()`, we need to actually use the provider:

```
private void initList() {
  if (model!=null) {
    stopManagingCursor(model);
    model.close();
  }

  String where=null;

  if (Intent.ACTION_SEARCH.equals(getIntent().getAction())) {
    where="name LIKE \"%"+getIntent().getStringExtra(SearchManager.QUERY)+"%\"";
  }

  model=managedQuery(RestaurantProvider.Columns.CONTENT_URI,
                     RestaurantProvider.DEFAULT_PROJECTION
                       .keySet()
                       .toArray(new String[0]),
                     where, null,
                     prefs.getString("sort_order", ""));

  adapter=new RestaurantAdapter(model);
  list.setAdapter(adapter);
}
```

We use the `DEFAULT_PROJECTION` from the `RestaurantProvider` to specify our list of columns to return. A safer implementation would have us spell out the specific columns, using the constants defined in `RestaurantProvider.Columns` (e.g., `NAME`).

Similarly, in `RestaurantWrapper`, while we can get by with our current column names (since `RestaurantProvider` uses the same names as does the underlying SQLite database), it would be safer to replace all the `String` constants (e.g., `"name"`) with references to the provider's constants (e.g., `NAME`).

Regardless, you can now rebuild and reinstall your application, which should run as it did before, just by using `RestaurantProvider` rather than having `LunchList` directly access the database.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Completely remove the `Restaurant` model class and have `LunchList` use the content provider for all CRUD operations.
- Add support for deleting restaurants from the system, by using the delete operation on the content provider.
- Create a separate application, perhaps cloned from `LunchList`, that uses the original `LunchList` `Restaurant` content provider from its own process.

TUTORIAL 39

Getting More From Your Contacts

In an earlier tutorial, we saw how to highlight certain timeline entries that were flagged as contacts. In this tutorial, we will extend this notion a bit further, creating our own synthetic `Cursor` that blends our friends list with the contacts data, so we can highlight our friends who are contacts in the friends list.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `37-UsingCP` edition of `Patchy` to use as a starting point.

Step #1: Create the FriendsCursor

If you have data that is more complicated than a simple array, but that you want to expose via a `Cursor` interface, `MatrixCursor` is a handy class. You can teach it the name of columns and feed it data, column by column within each row. The `MatrixCursor` then handles all of the `Cursor` API for you.

So, let us build a `FriendsCursor` subclass of `MatrixCursor` – create `Patchy/src/apt/tutorial/two/FriendsCursor.java` with the following content:

```
package apt.tutorial.two;

import android.app.Activity;
import android.database.MatrixCursor;
import android.util.Log;
import java.util.ArrayList;
import java.util.Collections;
import winterwell.jtwitter.Twitter;

public class FriendsCursor extends MatrixCursor {
  private static final String[] COLUMNS={"screenName", "_id"};

  FriendsCursor() {
    super(COLUMNS);
  }

  void populate(Twitter client, Activity ctxt) {
    final ArrayList<String> screenNames=new ArrayList<String>();

    try {
      for (Twitter.User u : client.getFriends()) {
        screenNames.add(u.screenName);
      }
    }
    catch (Throwable t) {
      Log.e("FriendsCursor",
            "Exception in JTwitter#getFriends()", t);
    }

    Collections.sort(screenNames);

    ctxt.runOnUiThread(new Runnable() {
      public void run() {
        int i=0;

        for (String u : screenNames) {
          newRow().add(u).add(i++);
        }
      }
    });
  };
}
```

Here, we do two things:

1. We initialize our list of columns, presently just the friend's screen name and the obligatory `_id` column
2. We add a `populate()` method, designed to be run on a background thread, that makes the Twitter API call to get our friends, then populates our `MatrixCursor`'s rows on the UI thread

Step #2: Use and Populate the FriendsCursor

Now that we have our `FriendsCursor`, we need to use it in `Patchy`.

Remove the code from `onCreate()` in `Patchy` that created and populated the `friends` `ArrayList` and corresponding `ArrayAdapter` with the following:

```
friends=new FriendsCursor();
```

You will need to change `friends` to be a `FriendsCursor` in the declaration of the data member, as well.

Then, add this snippet to `onCreate()` in `Patchy`, sometime after the code you have above, to populate our `FriendsCursor` in the background:

```
new Thread(new Runnable() {
  public void run() {
    friends.populate(getClient(), Patchy.this);
  }
}).start();
```

Step #3: Merge In Contact Information

Back in our `FriendsCursor`, at the moment, we are doing nothing to blend in data from the `Contacts` content provider. What we need to do is, in `populate()`, make appropriate content provider requests to see which of our friends is, indeed, a contact.

Modify `FriendsCursor` to look like the following:

```
package apt.tutorial.two;

import android.app.Activity;
import android.database.Cursor;
import android.database.MatrixCursor;
import android.provider.Contacts;
import android.util.Log;
import java.util.ArrayList;
import java.util.Collections;
import winterwell.jtwitter.Twitter;
```

```
public class FriendsCursor extends MatrixCursor {
  private static final String[] COLUMNS={"screenName",
                                         "isContact",
                                         "_id"};
  private static final String[] PROJECTION=new String[] {
                                        Contacts.People._ID,
                                                    };

  FriendsCursor() {
    super(COLUMNS);
  }

  void populate(Twitter client, final Activity ctxt) {
    final ArrayList<String> screenNames=new ArrayList<String>();

    try {
      for (Twitter.User u : client.getFriends()) {
        screenNames.add(u.screenName);
      }
    }
    catch (Throwable t) {
      Log.e("FriendsCursor",
            "Exception in JTwitter#getFriends()", t);
    }

    Collections.sort(screenNames);

    ctxt.runOnUiThread(new Runnable() {
      public void run() {
        int i=0;

        for (String u : screenNames) {
          int isContact=0;
          String[] args={u};
          StringBuffer query=new
StringBuffer(Contacts.OrganizationColumns.LABEL);

          query.append("='Twitter' AND ");
          query.append(Contacts.OrganizationColumns.COMPANY);
          query.append("=?");

          Cursor c=ctxt.managedQuery(Contacts.Organizations.CONTENT_URI,
                                     PROJECTION,
                                     query.toString(),
                                     args,
                                     Contacts.People.DEFAULT_SORT_ORDER);

          if (c.getCount()>0) {
            isContact=1;
          }

          c.close();

          newRow().add(u).add(isContact).add(i++);
```

```
        }
      }
    });
  };
}
```

This is not very efficient for long lists, as it makes individual queries per friend. A more efficient solution, perhaps using the `IN` operator in the `WHERE` clause, is left as an exercise for the reader.

At this point, though, each row in `FriendsCursor` tells us the friend's screen name and whether the friend is a contact.

Step #4: Highlight Contacts in Friends List

Finally, we need to tie our `FriendsCursor` into our `ListView` in `Patchy`. To do this, we could use a `SimpleCursorAdapter`, except that we would have no way to highlight which friends are contacts.

So, instead, we will subclass `SimpleCursorAdapter` and override `bindView()` to apply the highlights as needed.

Add the following `FriendsAdapter` implementation as an inner class to `Patchy`:

```
class FriendsAdapter extends SimpleCursorAdapter {
  FriendsAdapter() {
    super(Patchy.this,
          android.R.layout.simple_list_item_multiple_choice,
          friends, FRIENDS_PROJECTION, FRIENDS_WIDGETS);
  }

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

    if (cursor.getInt(1)==0) {
      view.setBackgroundColor(0x00000000);
    }
    else {
      view.setBackgroundColor(0xFF0000FF);
    }
```

```
  }
}
```

The `cursor.getInt(1)` code retrieves the second column from our `Cursor`, which happens to be our `isContact` value. To be safer, we could ask the `Cursor` which column is name `isContact`.

We also need to add a couple of static data members:

```
private static final String[] FRIENDS_PROJECTION={"screenName"};
private static final int[] FRIENDS_WIDGETS={android.R.id.text1};
```

Finally, we need to tie the `FriendsAdapter` into the `ListView`, up in `onCreate()`:

```
friendsAdapter=new FriendsAdapter();
friendsList.setAdapter(friendsAdapter);
```

At this point, rebuild and reinstall `Patchy`. Your friends who are also contacts should show up in the Friends tab highlighted in blue:

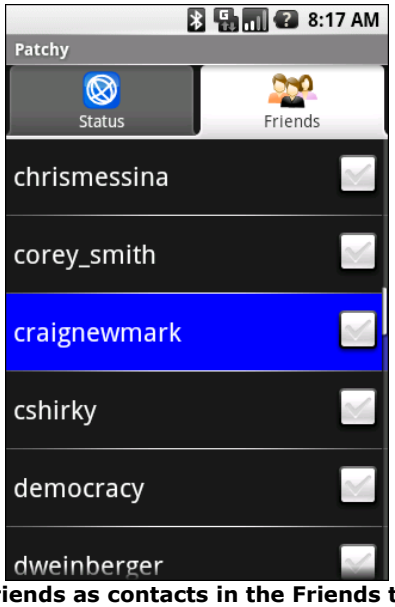


Figure 67. Friends as contacts in the Friends tab of Patchy

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Do a better job with the data model, such that we only need to look up data in the `Contacts` content provider from one place, rather than two (highlighting the timeline and highlighting friends).
- Add a context menu to the friends list, where you call or email friends who are also contacts. Also support direct-messaging those friends who subscribed to you.

Android Would Like Your Attention

From time to time, Android can tell you things that may be of interest, such as when the battery gets low. This allows you to take action based on those system events, such as reducing the amount of background work you do while the battery is low. In this tutorial, we will update the `TMonitor` service to be gentler while the battery is low, by polling less frequently.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `39-AdvCP` edition of `Patchy` to use as a starting point.

Step #1: Track the Battery State

We need to register a `BroadcastListener` from `TwitterMonitor` to be informed of changes in state in the battery.

First, add the `registerReceiver()` call to `onCreate()` in `TwitterMonitor`:

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

```
  super.onCreate();

  mgr=(NotificationManager)getSystemService(NOTIFICATION_SERVICE);
  new Thread(threadBody).start();
  registerReceiver(onBatteryChanged,
                   new IntentFilter(Intent.ACTION_BATTERY_CHANGED));
}
```

We set the receiver's `IntentFilter` to watch for `ACTION_BATTERY_CHANGED` events.

Next, call `unregisterReceiver()` in `onDestroy()` in `TwitterMonitor`:

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

  unregisterReceiver(onBatteryChanged);
  active.set(false);
}
```

If we do not do this, our service will keep running and getting battery events even after it would ordinarily have shut down.

Both of these rely on an `onBattery BroadcastReceiver` implementation, which you should add to `TwitterMonitor`:

```
BroadcastReceiver onBatteryChanged=new BroadcastReceiver() {
  public void onReceive(Context context, Intent intent) {
    int pct=100*intent.getIntExtra("level", 1)/intent.getIntExtra("scale", 1);

    isBatteryLow.set(pct<=25);
  }
};
```

All we do is examine the battery level (as measured on the battery level scale) and, if it hits 25% or lower, we set an `isBatteryLow` object to true; otherwise, we set it to false.

The `isBatteryLow` object is actually an `AtomicBoolean`, which you should add to the data members for `TwitterMonitor`:

```
private AtomicBoolean isBatteryLow=new AtomicBoolean(false);
```

At this point, we are watching for battery events and setting `isBatteryLow`, but we are not making use of that information anywhere.

Step #2: Use the Battery State

Then, all we need to do is look at `isBatteryLow` in our polling loop. Modify `threadBody` in `TwitterMonitor` to look like this:

```
private Runnable threadBody=new Runnable() {
  public void run() {
    while (active.get()) {
      for (Account l : accounts.values()) {
        poll(l);
        pollPeriod=POLL_PERIOD;

        if (isBatteryLow.get()) {
          pollPeriod*=10;
        }
      }

      SystemClock.sleep(pollPeriod);
    }
  }
};
```

All we do is multiply our normal polling period by a factor of `10` when the battery is low.

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Offer a user preference whereby `TwitterMonitor` will only poll if there is a WiFi connection available. Then, monitor the WiFi connection state and enable/disable polling as appropriate.
- Offer a user preference whereby `TwitterMonitor` will start collecting timeline updates on boot and will buffer some number of updates, so when `Patchy` connects, updates are available immediately and fewer are missed in between `Patchy` runs.

TUTORIAL 41

Now, Your Friends Are Alarmed

One current flaw with `Patchy` is that if your device goes to sleep, you will not get status updates on your timeline. Clearly, this must be corrected.

In this tutorial you will have `TwitterMonitor` use `AlarmManager`, instead of its own polling loop, to wake it up as needed. You will also use a `WakeLock` or two to ensure `TwitterMonitor` stays awake long enough to do a poll of the Twitter timeline.

Step-By-Step Instructions

First, you need to have completed the previous tutorial. If you are starting from scratch here, or if you wish to not use your existing work, you can download a ZIP file with all of the tutorial results, and you can copy the `40-SysEvents` edition of `Patchy` to use as a starting point.

Step #1: Remove the Background Thread

Since we are switching to using `AlarmManager`, we no longer need or want our background thread.

Convert the current definition of `threadBody`, the `Runnable` used by the background thread, into a `pollAll()` method in `TwitterMonitor`:

```
private void pollAll() {
  for (Account l : accounts.values()) {
    poll(l);
  }
}
```

Next, get rid of the `isActive` data member. Then, get rid of all references to `isActive` or `threadBody`, which you will find in `onCreate()` and `onDestroy()`.

Step #2: Define a Work Queue

We could, on every alarm, fork a thread to go do a `pollAll()`. However, if there are problems in the polling process, we might wind up in a situation where we keep forking new threads every minute, eventually crashing the `TwitterMonitor` process.

It is safer to have each alarm post an event to a work queue. That way, we can be assured of having only one background thread at any point in time. However, the work queue cannot merely be a `LinkedBlockingQueue`, as we need to make sure the device stays awake while there is work to be done.

With that in mind, create `TMonitor/src/apt/tutorial/three/WorkQueue.java` with the following implementation:

```
package apt.tutorial.three;

import android.content.Context;
import android.os.PowerManager;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.atomic.AtomicBoolean;

public class WorkQueue {
  private PowerManager.WakeLock lock=null;
  private BlockingQueue<Runnable> workQueue=new LinkedBlockingQueue<Runnable>();
  private AtomicBoolean isStopping=new AtomicBoolean(false);
  private Runnable onStop=null;

  public WorkQueue(Context context, String name,
                   Runnable onStop) {
    PowerManager
mgr=(PowerManager)context.getSystemService(Context.POWER_SERVICE);

    lock=mgr.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
```

```
                          name);
    lock.setReferenceCounted(true);
    this.onStop=onStop;
  }

  /*
  * Must be called within an outer active WakeLock!
  */
  public void enqueue(Runnable r) {
    synchronized(this) {
      if (!isStopping.get()) {
        workQueue.add(r);
        lock.acquire();
      }
    }
  }

  /*
  * Must be called within an outer active WakeLock!
  */
  public void stop() {
    synchronized(this) {
      isStopping.set(true);
      workQueue.add(onStop);
      lock.acquire();
    }
  }

  public void start() {
    new Thread(new Runnable() {
      public void run() {
        process();
      }
    }).start();
  }

  /*
  * Designed to run on its own thread!
  */
  private void process() {
    try {
      while (true) {
        Runnable r=workQueue.take();

        r.run();

        if (r==onStop) {
          break;
        }

        synchronized(this) {
          lock.release();
        }
      }
```

```
    }
    catch (InterruptedException e) {
      synchronized(this) {
        // theory = we really want to wrap up, so we
        // try to force the WakeLock to fully release
        // when we get interrupted by forces unknown
        lock.setReferenceCounted(false);
      }
    }

    synchronized(this) {
      lock.release();    // from the onStop job
    }
  }
}
```

Here, we wrap a `LinkedBlockingQueue` with our own API and manage a `WakeLock` to keep the device CPU running so long as we have work to be done. Events put on the queue are `Runnable` objects – `WorkQueue` just runs those `Runnable` instances as they get popped off the queue.

Step #3: Create the Service WakeLock

In addition to `WorkQueue` having a `WakeLock`, it is useful to have a separate `WakeLock` for use by the service. Specifically, when alarms go off, we need to ensure the service stays awake long enough to enqueue a piece of work. Then, the service no longer needs to be awake, and it is up to the `WorkQueue` to keep the CPU alive as long as is needed.

However, when the alarm goes off, the alarm `Intent` goes to a `BroadcastReceiver` we will set up later in this tutorial. Hence, we need the service's `WakeLock` to be available not only to the service itself, but to this `BroadcastReceiver`.

The easiest way to do that is via a static data member. Add a private static `PowerManager.WakeLock` to `TwitterMonitor`. Then, create a public lazy-getter for it:

```
synchronized public static PowerManager.WakeLock getLock(Context context) {
  if (lock==null) {
    PowerManager
mgr=(PowerManager)context.getSystemService(Context.POWER_SERVICE);
```

```
    lock=mgr.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
                         "apt.tutorial.three.TMonitorLock");
    lock.setReferenceCounted(true);
  }

  return(lock);
}
```

Step #4: Create the Alarm BroadcastReceiver

In order to receive alarms from the `AlarmManager`, we need a `BroadcastReceiver` registered in our `AndroidManifest.xml` file. This `BroadcastReceiver`, in turn, needs to start our `TwitterMonitor` service, in case it is not running, and to let it know that a poll is required.

Create `TMonitor/src/apt/tutorial/three/OnAlarmReceiver.java` with the following implementation:

```
package apt.tutorial.three;

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) {
    TwitterMonitor.getLock(context).acquire();

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

Then, add it to `AndroidManifest.xml`, along with the `WAKE_LOCK` permission we need in order to use a `WakeLock`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="apt.tutorial.three"
      android:versionCode="1"
      android:versionName="1.0">
  <uses-permission android:name="android.permission.INTERNET" />
  <uses-permission android:name="android.permission.WAKE_LOCK" />
```

```
  <application android:label="@string/app_name">
    <service android:name=".TwitterMonitor">
      <intent-filter>
        <action android:name="apt.tutorial.ITwitterMonitor" />
      </intent-filter>
    </service>
    <receiver android:name=".OnAlarmReceiver">
    </receiver>
  </application>
</manifest>
```

Note that `OnAlarmReceiver` acquires a `WakeLock` but does not release it – it will be released by the `TwitterMonitor` service in `onStart()`, which we will add later in this tutorial.

Step #5: Set Up the Work Queue

While we created `WorkQueue` earlier, we are not using it as yet. It is time to correct this oversight.

First, create a private `WorkQueue` data member in `TwitterMonitor`. Then, initialize it in `onCreate()`:

```
q=new WorkQueue(this, "apt.tutorial.three.TMonitorLock2",
                onStop);
q.start();
```

The `WorkQueue` constructor requires a `Runnable` that will be invoked when the queue is stopped, so add an `onStop` `Runnable` to `TwitterMonitor`:

```
private Runnable onStop=new Runnable() {
  public void run() {
    // being destroyed, nothing useful to do
  }
};
```

Also, stop the `WorkQueue` in `onDestroy()`:

```
q.stop();
```

The net effect is that our service will still only live while there is at least one client running (e.g., `Patchy`), but while it is running, it will have an operational `WorkQueue`.

Step #6: Enqueue the Work, Set the Alarm

Our `OnAlarmReceiver` calls `startService()` when an alarm goes off. Our service needs to implement `onStart()` and enqueue whatever work is supposed to be done.

So, add the following implementation of `onStart()` to `TwitterMonitor`:

```
@Override
public void onStart(Intent intent, final int startId) {
  enqueueJob();
  getLock(this).release();
}
```

We also need `enqueueJob()`:

```
private void enqueueJob() {
  q.enqueue(new Runnable() {
    public void run() {
      pollAll();
    }
  });
}
```

All we do is post a new `Runnable` to our `WorkQueue` that does a `pollAll()`, then release our `WakeLock`. The CPU should remain running, however, via the `WakeLock` managed by our `WorkQueue`.

We also need to augment `pollAll()` to set our alarm for the next poll, paying attention to our battery state:

```
private void pollAll() {
  for (Account l : accounts.values()) {
    poll(l);
  }

  setAlarm(isBatteryLow.get() ? POLL_PERIOD*10 : POLL_PERIOD);
}
```

In `onCreate()`, we need to set our initial alarm, to occur after a second or so:

```
setAlarm(INITIAL_POLL_PERIOD);
```

And, in `onDestroy()`, we should cancel any existing alarm, since we are shutting down:

```
alarm.cancel(pi);
```

Now, if you rebuild and reinstall the project, `Patchy` should work as before. The exception is that if you have `Patchy` and `TwitterMonitor` on a device, and you let the device fall asleep while `Patchy` is still active, you should still receive status updates.

Here is the entire listing of `TwitterMonitor` after all of these changes:

```
package apt.tutorial.three;

import android.app.AlarmManager;
import android.app.Notification;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.app.Service;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.IBinder;
import android.os.PowerManager;
import android.os.SystemClock;
import android.util.Log;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.atomic.AtomicBoolean;
import winterwell.jtwitter.Twitter;
import apt.tutorial.ITwitterListener;
import apt.tutorial.ITwitterMonitor;

public class TwitterMonitor extends Service {
  public static final String STATUS_UPDATE="apt.tutorial.three.STATUS_UPDATE";
  public static final String FRIEND="apt.tutorial.three.FRIEND";
  public static final String STATUS="apt.tutorial.three.STATUS";
```

```
  public static final String CREATED_AT="apt.tutorial.three.CREATED_AT";
  private static final int NOTIFY_ME_ID=1337;
  private static final int POLL_PERIOD=60000;
  private static final int INITIAL_POLL_PERIOD=1000;
  private static PowerManager.WakeLock lock=null;
  private Set<Long> seenStatus=new HashSet<Long>();
  private Map<ITwitterListener, Account> accounts=
          new ConcurrentHashMap<ITwitterListener, Account>();
  private List<String> bff=new CopyOnWriteArrayList<String>();
  private NotificationManager mgr=null;
  private AtomicBoolean isBatteryLow=new AtomicBoolean(false);
  private WorkQueue q=null;
  private AlarmManager alarm=null;
  private PendingIntent pi=null;
  private final ITwitterMonitor.Stub binder=new ITwitterMonitor.Stub() {
    public void registerAccount(String user, String password,
                                ITwitterListener callback) {
      registerAccountImpl(user, password, callback);
    }

    public void removeAccount(ITwitterListener callback) {
      removeAccountImpl(callback);
    }

    public void setBestFriends(ITwitterListener callback,
                               List<String> newBff) {
      bff.clear();
      bff.addAll(newBff);
    }

    public void updateTimeline() {
      new Thread(new Runnable() {
        public void run() {
          for (Account l : accounts.values()) {
            poll(l);
          }
        }
      }).start();
    }
  };

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

      lock=mgr.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
                           "apt.tutorial.three.TMonitorLock");
      lock.setReferenceCounted(true);
    }

    return(lock);
  }
```

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

  alarm=(AlarmManager)getSystemService(Context.ALARM_SERVICE);

  Intent i=new Intent(this, OnAlarmReceiver.class);

  pi=PendingIntent.getBroadcast(this, 0, i, 0);

  q=new WorkQueue(this, "apt.tutorial.three.TMonitorLock2",
                  onStop);
  q.start();

  mgr=(NotificationManager)getSystemService(NOTIFICATION_SERVICE);
  registerReceiver(onBatteryChanged,
                   new IntentFilter(Intent.ACTION_BATTERY_CHANGED));

  setAlarm(INITIAL_POLL_PERIOD);
}

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

@Override
public void onStart(Intent intent, final int startId) {
  enqueueJob();
  getLock(this).release();
}

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

  unregisterReceiver(onBatteryChanged);

  alarm.cancel(pi);
  q.stop();
}

public void registerAccountImpl(String user, String password,
                                ITwitterListener callback) {
  Account l=new Account(user, password, callback);

  accounts.put(callback, l);
}

public void removeAccountImpl(ITwitterListener callback) {
  accounts.remove(callback);
}

synchronized private void poll(Account l) {
```

```
    try {
      Twitter client=new Twitter(l.user, l.password);
      List<Twitter.Status> timeline=client.getFriendsTimeline();

      for (Twitter.Status s : timeline) {
        if (!seenStatus.contains(s.id)) {
          try {
            Intent broadcast=new Intent(STATUS_UPDATE);

            broadcast.putExtra(FRIEND, s.user.screenName);
            broadcast.putExtra(STATUS, s.text);
            broadcast.putExtra(CREATED_AT,
                               s.createdAt.toString());

            sendBroadcast(broadcast);

            seenStatus.add(s.id);
          }
          catch (Throwable t) {
            Log.e("TwitterMonitor", "Exception in callback", t);
          }

          if (bff.contains(s.user.screenName)) {
            notify(s.user.screenName);
          }
        }
      }
    }
    catch (Throwable t) {
      Log.e("TwitterMonitor", "Exception in poll()", t);
    }
  }

  private void notify(String friend) {
    Notification note=new Notification(R.drawable.red_ball,
                                       "Tweet!",
                                       System.currentTimeMillis());
    Intent i=new Intent();

    i.setClassName("apt.tutorial.two",
                   "apt.tutorial.two.Patchy");
    i.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);

    PendingIntent pi=PendingIntent.getActivity(this, 0, i,
                                               0);

    note.setLatestEventInfo(this, "Tweet!",
                            friend+" updated their Twitter status",
                            pi);

    mgr.notify(NOTIFY_ME_ID, note);
  }

  private void pollAll() {
```

```
    for (Account l : accounts.values()) {
      poll(l);
    }

    setAlarm(isBatteryLow.get() ? POLL_PERIOD*10 : POLL_PERIOD);
  }

  private void setAlarm(long period) {
    alarm.set(AlarmManager.ELAPSED_REALTIME_WAKEUP,
              SystemClock.elapsedRealtime()+period,
              pi);
  }

  private void enqueueJob() {
    q.enqueue(new Runnable() {
      public void run() {
        pollAll();
      }
    });
  }

  BroadcastReceiver onBatteryChanged=new BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
      int pct=100*intent.getIntExtra("level", 1)/intent.getIntExtra("scale", 1);

      isBatteryLow.set(pct<=25);
    }
  };

  private Runnable onStop=new Runnable() {
    public void run() {
      // being destroyed, nothing useful to do
    }
  };

  class Account {
    String user=null;
    String password=null;
    ITwitterListener callback=null;

    Account(String user, String password,
            ITwitterListener callback) {
      this.user=user;
      this.password=password;
      this.callback=callback;
    }
  }
}
```

Extra Credit

Here are some things you can try beyond those step-by-step instructions:

- Offer a user preference whereby `TwitterMonitor` will start collecting timeline updates on boot and will buffer some number of updates, so when `Patchy` connects, updates are available immediately and fewer are missed in between `Patchy` runs. This will require the initial alarm to be set from the on-boot receiver, rather than from the service's `onCreate()`.
- Extend the above extra credit component by offering a user preference to have `TwitterMonitor` raise a `Notification` if certain sorts of status updates are received (e.g., BFF) while `Patchy` itself is not running.

Keyword Index

Method.....