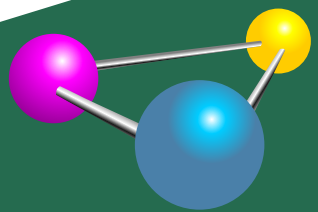
Version 2.0

Supports the Android 1.0 SDK

The Busy Coder's Guide to Android Development

Mark L. Murphy



CommonsWare

The Busy Coder's Guide to Android Development

by Mark L. Murphy

The Busy Coder's Guide to Android Development
by Mark L. Murphy

Printing History:
Jan 2009: Version 2.0 ISBN: 978-0-9816780-0-9

Table of Contents

Welcome to the Warescription!

We hope you enjoy this ebook and its updates – subscribe to the Warescription newsletter on the Warescription site to learn when new editions of this book, or other books, are available.

All editions of CommonsWare titles, print and ebook, follow a software-style numbering system. Major releases (1.0, 2.0, etc.) are available in both print and ebook; minor releases (0.1, 0.9, etc.) are available in ebook form for Warescription subscribers only. Releases ending in .9 are "release candidates" for the next major release, lacking perhaps an index but otherwise being complete.

Each Warescription ebook 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!

Thanks!

Thanks for your interest in developing applications for Android! Increasingly, people will access Internet-based services using so-called "non-traditional" means, such as mobile devices. The more we do in that space now, the more that people will help invest in that space to make it easier to build more powerful mobile applications in the future. Android is new – Android-powered devices appeared on the scene first in late 2008 – but it likely will rapidly grow in importance due to the size and scope of the Open Handset Alliance.

And, most of all, thanks for your interest in this book! I sincerely hope you find it useful and at least occasionally entertaining.

Prerequisites

If you are interested in programming for Android, you will need at least basic understanding of how to program in Java. Android programming is done using Java syntax, plus a class library that resembles a subset of the Java SE library (plus Android-specific extensions). If you have not programmed in Java before, you probably should learn how that works before attempting to dive into programming for Android.

The book does not cover in any detail how to download or install the Android development tools, either the Eclipse IDE flavor or the standalone flavor. The Android Web site covers this quite nicely. The material in the book should be relevant whether you use the IDE or not. You should download, install, and test out the Android development tools from the Android Web site before trying any of the examples listed in this book.

Some chapters may reference material in previous chapters, though usually with a link back to the preceding section of relevance.

Warescription

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

The Warescription entitles you, for the duration of your subscription, to ebook forms of *all* CommonsWare titles, not just the one you are reading. Presently, CommonsWare offers PDF and Kindle; other ebook 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 ebook form. That way, your ebooks 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, both short articles and 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 discount off the Warescription price.

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.

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 – just choose the tab of the book version you want, and click on the Source Code link for that tab. 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 **December 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.

Acknowledgments

I would like to thank the Android team, not only for putting out a good product, but for invaluable assistance on the Android Google Groups. In particular, I would like to thank Romain Guy, Justin@Google, and hackbod.

Icons used in the sample code were provided by the Nuvola icon set.

PART I – Core Concepts

CHAPTER 1

The Big Picture

Android devices, by and large, will be mobile phones. While the Android technology is being discussed for use in other areas (e.g., car dashboard "PCs"), for the most part, you can think of Android as being used on phones.

For developers, this has benefits and drawbacks.

On the plus side, circa 2008, Android-style smartphones are sexy. Offering Internet services over mobile devices dates back to the mid-1990's and the Handheld Device Markup Language (HDML). However, only in recent years have phones capable of Internet access taken off. Now, thanks to trends like text messaging and to products like Apple's iPhone, phones that can serve as Internet access devices are rapidly gaining popularity. So, working on Android applications gives you experience with an interesting technology (Android) in a fast-moving market segment (Internet-enabled phones), which is always a good thing.

The problem comes when you actually have to program the darn things.

Anyone with experience in programming for PDAs or phones has felt the pain of phones simply being *small* in all sorts of dimensions:

- Screens are small (you won't get comments like, "is that a 24-inch LCD in your pocket, or...?")
- Keyboards, if they exist, are small

- Pointing devices, if they exist, are annoying (as anyone who has lost their stylus will tell you) or inexact (large fingers and "multi-touch" LCDs are not a good mix)
- CPU speed and memory are tight compared to desktops and servers you may be used to
- You can have any programming language and development framework you want, so long as it was what the device manufacturer chose and burned into the phone's silicon
- And so on

Moreover, applications running on a phone have to deal with the fact that they're *on a phone*.

People with mobile phones tend to get very irritated when those phones don't work, which is why the "can you hear me now?" ad campaign from Verizon Wireless has been popular for the past few years. Similarly, those same people will get irritated at you if your program "breaks" their phone:

- ...by tying up the CPU such that calls can't be received
- ...by not working properly with the rest of the phone's OS, such that your application doesn't quietly fade to the background when a call comes in or needs to be placed
- ...by crashing the phone's operating system, such as by leaking memory like a sieve

Hence, developing programs for a phone is a different experience than developing desktop applications, Web sites, or back-end server processes. You wind up with different-looking tools, different-behaving frameworks, and "different than you're used to" limitations on what you can do with your program.

What Android tries to do is meet you halfway:

- You get a commonly-used programming language (Java) with some commonly used libraries (e.g., some Apache Commons APIs), with support for tools you may be used to (Eclipse)

- You get a fairly rigid and uncommon framework in which your programs need to run so they can be "good citizens" on the phone and not interfere with other programs or the operation of the phone itself

As you might expect, much of this book deals with that framework and how you write programs that work within its confines and take advantage of its capabilities.

What Androids Are Made Of

When you write a desktop application, you are "master of your own domain". You launch your main window and any child windows – like dialog boxes – that are needed. From your standpoint, you are your own world, leveraging features supported by the operating system, but largely ignorant of any other program that may be running on the computer at the same time. If you do interact with other programs, it is typically through an API, such as using JDBC (or frameworks atop it) to communicate with MySQL or another database.

Android has similar concepts, but packaged differently, and structured to make phones more crash-resistant.

Activities

The building block of the user interface is the **activity**. You can think of an activity as being the Android analogue for the window or dialog in a desktop application.

While it is possible for activities to not have a user interface, most likely your "headless" code will be packaged in the form of content providers or services, described below.

Content Providers

Content providers provide a level of abstraction for any data stored on the device that is accessible by multiple applications. The Android development model encourages you to make your own data available to other applications, as well as your own – building a content provider lets you do that, while maintaining complete control over how your data gets accessed.

Intents

Intents are system messages, running around the inside of the device, notifying applications of various events, from hardware state changes (e.g., an SD card was inserted), to incoming data (e.g., an SMS message arrived), to application events (e.g., your activity was launched from the device's main menu). Not only can you respond to intents, but you can create your own, to launch other activities, or to let you know when specific situations arise (e.g., raise such-and-so intent when the user gets within 100 meters of this-and-such location).

Services

Activities, content providers, and intent receivers are all short-lived and can be shut down at any time. Services, on the other hand, are designed to keep running, if needed, independent of any activity. You might use a service for checking for updates to an RSS feed, or to play back music even if the controlling activity is no longer operating.

Stuff At Your Disposal

Storage

You can package data files with your application, for things that do not change, such as icons or help files. You also can carve out a small bit of space on the device itself, for databases or files containing user-entered or retrieved data needed by your application. And, if the user supplies bulk storage, like an SD card, you can read and write files on there as needed.

Network

Android devices will generally be Internet-ready, through one communications medium or another. You can take advantage of the Internet access at any level you wish, from raw Java sockets all the way up to a built-in WebKit-based Web browser widget you can embed in your application.

Multimedia

Android devices have the ability to play back and record audio and video. While the specifics may vary from device to device, you can query the device to learn its capabilities and then take advantage of the multimedia capabilities as you see fit, whether that is to play back music, take pictures with the camera, or use the microphone for audio note-taking.

GPS

Android devices will frequently have access to location providers, such as GPS, that can tell your applications where the device is on the face of the Earth. In turn, you can display maps or otherwise take advantage of the location data, such as tracking a device's movements if the device has been stolen.

Phone Services

And, of course, Android devices are typically phones, allowing your software to initiate calls, send and receive SMS messages, and everything else you expect from a modern bit of telephony technology.

CHAPTER 2

Project Structure

The Android build system is organized around a specific directory tree structure for your Android project, much like any other Java project. The specifics, though, are fairly unique to Android and what it all does to prepare the actual application that will run on the device or emulator. Here's a quick primer on the project structure, to help you make sense of it all, particularly for the sample code referenced in this book.

Root Contents

When you create a new Android project (e.g., via `activitycreator`), you get several items in the project's root directory:

- `AndroidManifest.xml`, which is an XML file describing the application being built and what components – activities, services, etc. – are being supplied by that application
- `build.xml`, which is an Ant script for compiling the application and installing it on the device
- `default.properties`, a property file used by the Ant build script
- `bin/`, which holds the application once it is compiled
- `libs/`, which holds any third-party Java JARs your application requires
- `src/`, which holds the Java source code for the application

CHAPTER 3

Inside the Manifest

The foundation for any Android application is the manifest file: `AndroidManifest.xml` in the root of your project. Here is where you declare what all is inside your application – the activities, the services, and so on. You also indicate how these pieces attach themselves to the overall Android system; for example, you indicate which activity (or activities) should appear on the device's main menu (a.k.a., launcher).

When you create your application, you will get a starter manifest generated for you. For a simple application, offering a single activity and nothing else, the auto-generated manifest will probably work out fine, or perhaps require a few minor modifications. On the other end of the spectrum, the manifest file for the Android API demo suite is over 1,000 lines long. Your production Android applications will probably fall somewhere in the middle.

Most of the interesting bits of the manifest will be described in greater detail in the chapters on their associated Android features. For example, the `service` element will be described in greater detail in the chapter on creating services. For now, we just need to understand what the role of the manifest is and its general overall construction.

In The Beginning, There Was the Root, And It Was Good

The root of all manifest files is, not surprisingly, a `manifest` element:

PART II – Activities

CHAPTER 4

Creating a Skeleton Application

Every programming language or environment book starts off with the ever-popular "Hello, World!" demonstration: just enough of a program to prove you can build things, not so much that you cannot understand what is going on. However, the typical "Hello, World!" program has no interactivity (e.g., just dumps the words to a console), and so is really boring.

This chapter demonstrates a simple project, but one using Advanced Push-Button Technology™ and the current time, to show you how a simple Android activity works.

Begin at the Beginning

To work with anything in Android, you need a project. With ordinary Java, if you wanted, you could just write a program as a single file, compile it with `javac`, and run it with `java`, without any other support structures. Android is more complex, but to help keep it manageable, Google has supplied tools to help create the project. If you are using an Android-enabled IDE, such as Eclipse with the Android plugin, you can create a project inside of the IDE (e.g., select **File** > **New** > **Project**, then choose **Android** > **Android Project**).

If you are using tools that are not Android-enabled, you can use the `activitycreator` script, found in the `tools/` directory in your SDK installation. Just pass `activitycreator` the package name of the activity you

want to create and a `--out` switch indicating where the project files should be generated. For example:

```
activitycreator --out /path/to/my/project/dir \
com.commonsware.android.Now
```

You will wind up with a handful of pre-generated files, as described in a previous chapter.

For the purposes of the samples shown in this book, you can download their project directories in a ZIP file on the CommonsWare Web site. These projects are ready for use; you do not need to run `activitycreator` on those unpacked samples.

The Activity

Your project's `src/` directory contains the standard Java-style tree of directories based upon the Java package you chose when you created the project (e.g., `com.commonsware.android` results in `src/com/commonsware/android/`). Inside the innermost directory you should find a pre-generated source file named `Now.java`, which is where your first activity will go.

Open `Now.java` in your editor and paste in the following code:

```
package com.commonsware.android.skeleton;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import java.util.Date;

public class Now extends Activity implements View.OnClickListener {
  Button btn;

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

    btn = new Button(this);
    btn.setOnClickListener(this);
```

```
    updateTime();
    setContentView(btn);
  }

  public void onClick(View view) {
    updateTime();
  }

  private void updateTime() {
    btn.setText(new Date().toString());
  }
}
```

Or, if you download the source files off the Web site, you can just use the `Skeleton/Now` project directly.

Dissecting the Activity

Let's examine this piece by piece:

```
package com.commonsware.android.skeleton;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import java.util.Date;
```

The package declaration needs to be the same as the one you used when creating the project. And, like any other Java project, you need to import any classes you reference. Most of the Android-specific classes are in the `android` package.

Remember that not every Java SE class is available to Android programs! Visit the Android class reference to see what is and is not available.

```
public class Now extends Activity implements View.OnClickListener {
  Button btn;
```

Activities are public classes, inheriting from the `android.Activity` base class. In this case, the activity holds a button (`btn`). Since, for simplicity, we want

to trap all button clicks just within the activity itself, we also have the activity class implement `OnClickListener`.

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

  btn = new Button(this);
  btn.setOnClickListener(this);
  updateTime();
  setContentView(btn);
}
```

The `onCreate()` method is invoked when the activity is started. The first thing you should do is chain upward to the superclass, so the stock Android activity initialization can be done.

In our implementation, we then create the button instance (`new Button(this)`), tell it to send all button clicks to the activity instance itself (via `setOnClickListener()`), call a private `updateTime()` method (see below), and then set the activity's content view to be the button itself (via `setContentView()`).

We will discuss that magical `Bundle icicle` in a later chapter. For the moment, consider it an opaque handle that all activities receive upon creation.

```
public void onClick(View view) {
  updateTime();
}
```

In Swing, a `JButton` click raises an `ActionEvent`, which is passed to the `ActionListener` configured for the button. In Android, a button click causes `onClick()` to be invoked in the `OnClickListener` instance configured for the button. The listener is provided the view that triggered the click (in this case, the button). All we do here is call that private `updateTime()` method:

```
private void updateTime() {
  btn.setText(new Date().toString());
}
```

When we open the activity (`onCreate()`) or when the button is clicked (`onClick()`), we update the button's label to be the current time via `setText()`, which functions much the same as the `JButton` equivalent.

Building and Running the Activity

To build the activity, either use your IDE's built-in Android packaging tool, or run ant in the base directory of your project. Then, to run the activity:

- Launch the emulator (e.g., run `tools/emulator` from your Android SDK installation)

Figure 1. The Android home screen

- Install the package (e.g., run `tools/adb install /path/to/this/example/bin/Now.apk` from your Android SDK installation)
- View the list of installed applications in the emulator and find the "Now" application

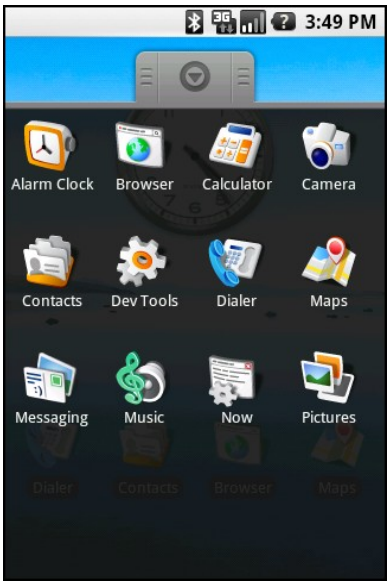


Figure 2. The Android application "launcher"

- Open that application

You should see an activity screen akin to:

Figure 3. The Now demonstration activity

Clicking the button – in other words, pretty much anywhere on the phone's screen – will update the time shown in the button's label.

Note that the label is centered horizontally and vertically, as those are the default styles applied to button captions. We can control that formatting, which will be covered in a later chapter.

After you are done gazing at the awesomeness of Advanced Push-Button Technology™, you can click the back button on the emulator to return to the launcher.

CHAPTER 5

Using XML-Based Layouts

While it is technically possible to create and attach widgets to our activity purely through Java code, the way we did in the preceding chapter, the more common approach is to use an XML-based layout file. Dynamic instantiation of widgets is reserved for more complicated scenarios, where the widgets are not known at compile-time (e.g., populating a column of radio buttons based on data retrieved off the Internet).

With that in mind, it's time to break out the XML and learn how to lay out Android activity views that way.

What Is an XML-Based Layout?

As the name suggests, an XML-based layout is a specification of widgets' relationships to each other – and to containers – encoded in XML format. Specifically, Android considers XML-based layouts to be resources, and as such layout files are stored in the `res/layout` directory inside your Android project.

Each XML file contains a tree of elements specifying a layout of widgets and containers that make up one `View`. The attributes of the XML elements are properties, describing how a widget should look or how a container should behave. For example, if a `Button` element has an attribute value of `android:textStyle = "bold"`, that means that the text appearing on the face of the button should be rendered in a boldface font style.

CHAPTER 6

Employing Basic Widgets

Every GUI toolkit has some basic widgets: fields, labels, buttons, etc. Android's toolkit is no different in scope, and the basic widgets will provide a good introduction as to how widgets work in Android activities.

Assigning Labels

The simplest widget is the label, referred to in Android as a `TextView`. Like in most GUI toolkits, labels are bits of text not editable directly by users. Typically, they are used to identify adjacent widgets (e.g., a "Name:" label before a field where one fills in a name).

In Java, you can create a label by creating a `TextView` instance. More commonly, though, you will create labels in XML layout files by adding a `TextView` element to the layout, with an `android:text` property to set the value of the label itself. If you need to swap labels based on certain criteria, such as internationalization, you may wish to use a resource reference in the XML instead, as will be described later in this book.

`TextView` has numerous other properties of relevance for labels, such as:

- `android:typeface` to set the typeface to use for the label (e.g., `monospace`)
- `android:textStyle` to indicate that the typeface should be made bold (`bold`), italic (`italic`), or bold and italic (`bold_italic`)

CHAPTER 7

Working with Containers

Containers pour a collection of widgets (and possibly child containers) into specific layouts you like. If you want a form with labels on the left and fields on the right, you will need a container. If you want OK and Cancel buttons to be beneath the rest of the form, next to one another, and flush to right side of the screen, you will need a container. Just from a pure XML perspective, if you have multiple widgets (beyond `RadioButton` widgets in a `RadioGroup`), you will need a container just to have a root element to place the widgets inside.

Most GUI toolkits have some notion of layout management, frequently organized into containers. In Java/Swing, for example, you have layout managers like `BoxLayout` and containers that use them (e.g., `Box`). Some toolkits stick strictly to the box model, such as XUL and Flex, figuring that any desired layout can be achieved through the right combination of nested boxes.

Android, through `LinearLayout`, also offers a "box" model, but in addition supports a range of containers providing different layout rules. In this chapter, we will look at three commonly-used containers: `LinearLayout` (the box model), `RelativeLayout` (a rule-based model), and `TableLayout` (the grid model), along with `ScrollView`, a container designed to assist with implementing scrolling containers. In the next chapter, we will examine some more esoteric containers.

CHAPTER 8

Using Selection Widgets

Back in the chapter on basic widgets, you saw how fields could have constraints placed upon them to limit possible input, such as numeric-only or phone-number-only. These sorts of constraints help users "get it right" when entering information, particularly on a mobile device with cramped keyboards.

Of course, the ultimate in constrained input is to select a choice from a set of items, such as the radio buttons seen earlier. Classic UI toolkits have listboxes, comboboxes, drop-down lists, and the like for that very purpose. Android has many of the same sorts of widgets, plus others of particular interest for mobile devices (e.g., the `Gallery` for examining saved photos).

Moreover, Android offers a flexible framework for determining what choices are available in these widgets. Specifically, Android offers a framework of data adapters that provide a common interface to selection lists ranging from static arrays to database contents. Selection views – widgets for presenting lists of choices – are handed an adapter to supply the actual choices.

Adapting to the Circumstances

In the abstract, adapters provide a common interface to multiple disparate APIs. More specifically, in Android's case, adapters provide a common interface to the data model behind a selection-style widget, such as a listbox.

CHAPTER 9

Getting Fancy With Lists

The humble `ListView` is one of the most important widgets in all of Android, simply because it is used so frequently. Whether choosing a contact to call or an email message to forward or an ebook to read, `ListView` widgets are employed in a wide range of activities.

Of course, it would be nice if they were more than just plain text.

The good news is that they can be as fancy as you want, within the limitations of a mobile device's screen, of course. However, making them fancy takes some work and some features of Android that we will cover in this chapter.

The material in this chapter is based on the author's posts to the Building 'Droids column on AndroidGuys.com.

Getting To First Base

The classic Android `ListView` is a plain list of text — solid but uninspiring. This is because all we have handed to the `ListView` is a bunch of words in an array, and told Android to use a simple built-in layout for pouring those words into a list.

CHAPTER 10

Employing Fancy Widgets and Containers

The widgets and containers covered to date are not only found in many GUI toolkits (in one form or fashion), but also are widely used in building GUI applications, whether Web-based, desktop, or mobile. The widgets and containers in this chapter are a little less widely used, though you will likely find many to be quite useful.

Pick and Choose

With limited-input devices like phones, having widgets and dialogs that are aware of the type of stuff somebody is supposed to be entering is very helpful. It minimizes keystrokes and screen taps, plus reduces the chance of making some sort of error (e.g., entering a letter someplace where only numbers are expected).

As shown previously, `EditText` has content-aware flavors for entering in numbers, phone numbers, etc. Android also supports widgets (`DatePicker`, `TimePicker`) and dialogs (`DatePickerDialog`, `TimePickerDialog`) for helping users enter dates and times.

The `DatePicker` and `DatePickerDialog` allow you to set the starting date for the selection, in the form of a year, month, and day of month value. Note that the month runs from `0` for January through `11` for December. Most

CHAPTER 11

Applying Menus

Like applications for the desktop and some mobile operating systems, such as PalmOS and Windows Mobile, Android supports activities with "application" menus. Some Android phones will have a dedicated menu key for popping up the menu; others will offer alternate means for triggering the menu to appear.

Also, as with many GUI toolkits, you can create "context menus". On a traditional GUI, this might be triggered by the right-mouse button. On mobile devices, context menus typically appear when the user "taps-and-holds" over a particular widget. For example, if a `TextView` had a context menu, and the device was designed for finger-based touch input, you could push the `TextView` with your finger, hold it for a second or two, and a pop-up menu will appear for the user to choose from.

Where Android differs from most other GUI toolkits is in terms of menu construction. While you can add items to the menu, you do not have full control over the menu's contents, nor the timing of when the menu is built. Part of the menu is system-defined, and that portion is managed by the Android framework itself.

Flavors of Menu

Android considers the two types of menu described above as being the "options menu" and "context menu". The options menu is triggered by

CHAPTER 12

Fonts

Inevitably, you'll get the question "hey, can we change this font?" when doing application development. The answer depends on what fonts come with the platform, whether you can add other fonts, and how to apply them to the widget or whatever needs the font change.

Android is no different. It comes with some fonts plus a means for adding new fonts. Though, as with any new environment, there are a few idiosyncrasies to deal with.

Love The One You're With

Android natively knows three fonts, by the shorthand names of "sans", "serif", and "monospace". These fonts are actually the Droid series of fonts, created for the Open Handset Alliance by Ascender.

For those fonts, you can just reference them in your layout XML, if you choose, such as the following layout from the `Fonts/FontSampler` sample project:

```
<?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>
```

CHAPTER 13

Embedding the WebKit Browser

Other GUI toolkits let you use HTML for presenting information, from limited HTML renderers (e.g., Java/Swing, wxWidgets) to embedding Internet Explorer into .NET applications. Android is much the same, in that you can embed the built-in Web browser as a widget in your own activities, for displaying HTML or full-fledged browsing. The Android browser is based on WebKit, the same engine that powers Apple's Safari Web browser.

The Android browser is sufficiently complex that it gets its own Java package (`android.webkit`), though using the `WebView` widget itself can be simple or powerful, based upon your requirements.

A Browser, Writ Small

For simple stuff, `WebView` is not significantly different than any other widget in Android – pop it into a layout, tell it what URL to navigate to via Java code, and you're done.

For example (`WebKit/Browser1`), here is a simple layout with a `WebView`:

```
<?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"
```

CHAPTER 14

Showing Pop-Up Messages

Sometimes, your activity (or other piece of Android code) will need to speak up.

Not every interaction with Android users will be neat, tidy, and containable in activities composed of views. Errors will crop up. Background tasks may take way longer than expected. Something asynchronous may occur, such as an incoming message. In these and other cases, you may need to communicate with the user outside the bounds of the traditional user interface.

Of course, this is nothing new. Error messages in the form of dialog boxes have been around for a very long time. More subtle indicators also exist, from task tray icons to bouncing dock icons to a vibrating cell phone.

Android has quite a few systems for letting you alert your users outside the bounds of an `Activity`-based UI. One, notifications, is tied heavily into intents and services and, as such, is covered in a later chapter. In this chapter, you will see two means of raising pop-up messages: toasts and alerts.

Raising Toasts

A `Toast` is a transient message, meaning that it displays and disappears on its own without user interaction. Moreover, it does not take focus away from

CHAPTER 15

Dealing with Threads

Ideally, you want your activities to be downright snappy, so your users don't feel that your application is sluggish. Responding to user input quickly (e.g., 200ms) is a fine goal. At minimum, though, you need to make sure you respond within 5 seconds, lest the `ActivityManager` decide to play the role of the Grim Reaper and kill off your activity as being non-responsive.

Of course, your activity might have real work to do, which takes non-negligible amounts of time. There are two ways of dealing with this:

1. Do expensive operations in a background service, relying on notifications to prompt users to go back to your activity
2. Do expensive work in a background thread

Android provides a veritable cornucopia of means to set up background threads yet allow them to safely interact with the UI on the UI thread. These include `Handler` objects and posting `Runnable` objects to the `View`.

Getting Through the Handlers

The most flexible means of making an Android-friendly background thread is to create an instance of a `Handler` subclass. You only need one `Handler` object per activity, and you do not need to manually register it or anything – merely creating the instance is sufficient to register it with the Android threading subsystem.

CHAPTER 16

Handling Activity Lifecycle Events

While this may sound like a broken record...please remember that Android devices, by and large, are phones. As such, some activities are more important that others – taking a call is probably more important to users than is playing Sudoku. And, since it is a phone, it probably has less RAM than does your current desktop or notebook.

As a result, your activity may find itself being killed off because other activities are going on and the system needs your activity's memory. Think of it as the Android equivalent of the "circle of life" – your activity dies so others may live, and so on. You cannot assume that your activity will run until you think it is complete, or even until the user thinks it is complete.

This is one example – perhaps the most important example – of how an activity's lifecycle will affect your own application logic. This chapter covers the various states and callbacks that make up an activity's lifecycle and how you can hook into them appropriately.

Schroedinger's Activity

An activity, generally speaking, is in one of four states at any point in time:

PART III – Data Stores, Network Services, and APIs

CHAPTER 17

Using Preferences

Android has many different ways for you to store data for long-term use by your activity. The simplest to use is the preferences system.

Android allows activities and applications to keep preferences, in the form of key/value pairs (akin to a `Map`), that will hang around between invocations of an activity. As the name suggests, the primary purpose is for you to store user-specified configuration details, such as the last feed the user looked at in your feed reader, or what sort order to use by default on a list, or whatever. Of course, you can store in the preferences whatever you like, so long as it is keyed by a `String` and has a primitive value (`boolean`, `String`, etc.)

Preferences can either be for a single activity or shared among all activities in an application. Eventually, preferences might be shareable across applications, but that is not supported as of the time of this writing.

Getting What You Want

To get access to the preferences, you have three APIs to choose from:

1. `getPreferences()` from within your `Activity`, to access activity-specific preferences
2. `getSharedPreferences()` from within your `Activity` (or other application `Context`), to access application-level preferences

CHAPTER 18

Accessing Files

While Android offers structured storage, via preferences and databases, sometimes a simple file will suffice. Android offers two models for accessing files: one for files pre-packaged with your application, and one for files created on-device by your application.

You And The Horse You Rode In On

Let's suppose you have some static data you want to ship with the application, such as a list of words for a spell-checker. The easiest way to deploy that is to put the file in the `res/raw` directory, so it gets put in the Android application `.apk` file as part of the packaging process as a raw resource.

To access this file, you need to get yourself a `Resources` object. From an activity, that is as simple as calling `getResources()`. A `Resources` object offers `openRawResource()` to get an `InputStream` on the file you specify. Rather than a path, `openRawResource()` expects an integer identifier for the file as packaged. This works just like accessing widgets via `findViewById()` – if you put a file named `words.xml` in `res/raw`, the identifier is accessible in Java as `R.raw.words`.

Since you can only get an `InputStream`, you have no means of modifying this file. Hence, it is really only useful for static reference data. Moreover, since it is unchanging until the user installs an updated version of your application

CHAPTER 19

Working with Resources

Resources are static bits of information held outside the Java source code. You have seen one type of resource – the layout – frequently in the examples in this book. There are many other types of resource, such as images and strings, that you can take advantage of in your Android applications.

The Resource Lineup

Resources are stored as files under the `res/` directory in your Android project layout. With the exception of raw resources (`res/raw/`), all the other types of resources are parsed for you, either by the Android packaging system or by the Android system on the device or emulator. So, for example, when you lay out an activity's UI via a layout resource (`res/layout/`), you do not have to parse the layout XML yourself – Android handles that for you.

In addition to layout resources (first seen in an earlier chapter) and raw resources (introduced in another earlier chapter), there are several other types of resource available to you, including:

- Animations (`res/anim/`), designed for short clips as part of a user interface, such as an animation suggesting the turning of a page when a button is clicked
- Images (`res/drawable`), for putting static icons or other pictures in a user interface

CHAPTER 20

Managing and Accessing Local Databases

SQLite is a very popular embedded database, as it combines a clean SQL interface with a very small memory footprint and decent speed. Moreover, it is public domain, so everyone can use it. Lots of firms (Adobe, Apple, Google, Sun, Symbian) and open source projects (Mozilla, PHP, Python) all ship products with SQLite.

For Android, SQLite is "baked into" the Android runtime, so every Android application can create SQLite databases. Since SQLite uses a SQL interface, it is fairly straightforward to use for people with experience in other SQL-based databases. However, its native API is not JDBC, and JDBC might be too much overhead for a memory-limited device like a phone, anyway. Hence, Android programmers have a different API to learn – the good news being is that it is not that difficult.

This chapter will cover the basics of SQLite use in the context of working on Android. It by no means is a thorough coverage of SQLite as a whole. If you want to learn more about SQLite and how to use it in other environment than Android, a fine book is The Definitive Guide to SQLite by Michael Owens.

Activities will typically access a database via a content provider or service. As such, this chapter does not have a full example. You will find a full example

CHAPTER 21

Leveraging Java Libraries

Java has as many, if not more, third-party libraries than any other modern programming language. Here, "third-party libraries" refer to the innumerable JARs that you can include in a server or desktop Java application – the things that the Java SDKs themselves do not provide.

In the case of Android, the Dalvik VM at its heart is not precisely Java, and what it provides in its SDK is not precisely the same as any traditional Java SDK. That being said, many Java third-party libraries still provide capabilities that Android lacks natively and therefore may be of use to you in your project, for the ones you can get working with Android's flavor of Java.

This chapter explains what it will take for you to leverage such libraries and the limitations on Android's support for arbitrary third-party code.

The Outer Limits

Not all available Java code, of course, will work well with Android. There are a number of factors to consider, including:

- **Expected Platform APIs**: Does the code assume a newer JVM than the one Android is based on? Or, does the code assume the existence of Java APIs that ship with J2SE but not with Android, such as Swing?

CHAPTER 22

Communicating via the Internet

The expectation is that most, if not all, Android devices will have built-in Internet access. That could be WiFi, cellular data services (EDGE, 3G, etc.), or possibly something else entirely. Regardless, most people – or at least those with a data plan or WiFi access – will be able to get to the Internet from their Android phone.

Not surprisingly, the Android platform gives developers a wide range of ways to make use of this Internet access. Some offer high-level access, such as the integrated WebKit browser component we saw in an earlier chapter. If you want, you can drop all the way down to using raw sockets. Or, in between, you can leverage APIs – both on-device and from 3rd-party JARs – that give you access to specific protocols: HTTP, XMPP, SMTP, and so on.

The emphasis of this book is on the higher-level forms of access: the WebKit component and Internet-access APIs, as busy coders should be trying to reuse existing components versus rolling one's own on-the-wire protocol wherever possible.

REST and Relaxation

Android does not have built-in SOAP or XML-RPC client APIs. However, it does have the Apache HttpComponents library baked in. You can either layer a SOAP/XML-RPC layer atop this library, or use it "straight" for accessing REST-style Web services. For the purposes of this book, "REST-

PART IV – Intents

CHAPTER 23

Creating Intent Filters

Up to now, the focus of this book has been on activities opened directly by the user from the device's launcher. This, of course, is the most obvious case for getting your activity up and visible to the user. And, in many cases it is the primary way the user will start using your application.

However, remember that the Android system is based upon lots of loosely-coupled components. What you might accomplish in a desktop GUI via dialog boxes, child windows, and the like are mostly supposed to be independent activities. While one activity will be "special", in that it shows up in the launcher, the other activities all need to be reached...somehow.

The "how" is via intents.

An intent is basically a message that you pass to Android saying, "Yo! I want to do...er...something! Yeah!" How specific the "something" is depends on the situation – sometimes you know exactly what you want to do (e.g., open up one of your other activities), and sometimes you don't.

In the abstract, Android is all about intents and receivers of those intents. So, now that we are well-versed in creating activities, let's dive into intents, so we can create more complex applications while simultaneously being "good Android citizens".

CHAPTER 24

Launching Activities and Sub-Activities

As discussed previously, the theory behind the Android UI architecture is that developers should decompose their application into distinct activities, each implemented as an `Activity`, each reachable via intents, with one "main" activity being the one launched by the Android launcher. For example, a calendar application could have activities for viewing the calendar, viewing a single event, editing an event (including adding a new one), and so forth.

This, of course, implies that one of your activities has the means to start up another activity. For example, if somebody clicks on an event from the view-calendar activity, you might want to show the view-event activity for that event. This means that, somehow, you need to be able to cause the view-event activity to launch and show a specific event (the one the user clicked upon).

This can be further broken down into two scenarios:

1. You know what activity you want to launch, probably because it is another activity in your own application
2. You have a content `Uri` to...something, and you want your users to be able to do...something with it, but you do not know up front what the options are

CHAPTER 25

Finding Available Actions via Introspection

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

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

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

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

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

CHAPTER 26

Handling Rotation

Some Android handsets, like the T-Mobile G1, offer a slide-out keyboard that triggers rotating the screen from portrait to landscape. Other handsets might use accelerometers to determine screen rotation, like the iPhone does. As a result, it is reasonable to assume that switching from portrait to landscape and back again may be something your users will look to do.

Android has a number of ways for you to handle screen rotation, so your application can properly handle either orientation. All these facilities do is help you detect and manage the rotation process – you are still required to make sure you have layouts that look decent on each orientation.

Much of the material in this chapter originally appeared in blog posts in the Building 'Droids column on AndroidGuys.

A Philosophy of Destruction

By default, when there is a change in the phone configuration that might affect resource selection, Android will destroy and re-create any running or paused activities the next time they are to be viewed. While this could happen for a variety of different configuration changes (e.g., change of language selection), it will most likely trip you up mostly for rotations, since a change in orientation can cause you to load a different set of resources (e.g., layouts).

The key here is that this is the default behavior. It may even be the behavior that is best for one or more of your activities. You do have some control over the matter, though, and can tailor how your activities respond to orientation changes or similar configuration switches.

It's All The Same, Just Different

Since, by default, Android destroys and re-creates your activity on a rotation, you may only need to hook into the same `onSaveInstanceState()` that you would if your activity were destroyed for any other reason (e.g., low memory). Implement that method in your activity and fill in the supplied `Bundle` with enough information to get you back to your current state. Then, in `onCreate()` (or `onRestoreInstanceState()`, if you prefer), pick the data out of the `Bundle` and use it to bring your activity back to the way it was.

To demonstrate this, let's take a look at the `Rotation/RotationOne` project. It, and the other sample projects used in this chapter, use a pair of main.xml layouts, one in `res/layout/` and one in `res/layout-land/` for use in landscape mode. Here is the portrait layout:

```
<?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"
  >
  <Button android:id="@+id/pick"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_weight="1"
    android:text="Pick"
    android:enabled="true"
  />
  <Button android:id="@+id/view"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_weight="1"
    android:text="View"
    android:enabled="false"
  />
</LinearLayout>
```

While here is the similar landscape layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
  <Button android:id="@+id/pick"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_weight="1"
    android:text="Pick"
    android:enabled="true"
  />
  <Button android:id="@+id/view"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_weight="1"
    android:text="View"
    android:enabled="false"
  />
</LinearLayout>
```

Basically, it is a pair of buttons, each taking up half the screen. In portrait mode, the buttons are stacked; in landscape mode, they are side-by-side.

If you were to simply create a project, put in those two layouts, and compile it, the application would appear to work just fine – a rotation (<Ctrl>-<F12> in the emulator) will cause the layout to change. And while buttons lack state, if you were using other widgets (e.g., `EditText`), you would even find that Android hangs onto some of the widget state for you (e.g., the text entered in the `EditText`).

What Android cannot automatically help you with is anything held outside the widgets.

This application is derived from the Pick demo used in an earlier chapter. There, clicking one button would let you pick a contact, then view the contact. Here, we split those into separate buttons, with the "View" button only enabled when we actually have a contact.

Let's see how we handle this, using `onSaveInstanceState()`:

```
public class RotationOneDemo extends Activity {
  static final int PICK_REQUEST=1337;
  Button viewButton=null;
  Uri contact=null;

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

    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("content://contacts/people"));

        startActivityForResult(i, PICK_REQUEST);
      }
    });

    viewButton=(Button)findViewById(R.id.view);

    viewButton.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        startActivity(new Intent(Intent.ACTION_VIEW, contact));
      }
    });

    restoreMe(savedInstanceState);

    viewButton.setEnabled(contact!=null);
  }

  @Override
  protected void onActivityResult(int requestCode, int resultCode,
                                  Intent data) {
    if (requestCode==PICK_REQUEST) {
      if (resultCode==RESULT_OK) {
        contact=data.getData();
        viewButton.setEnabled(true);
      }
    }
  }

  @Override
  protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);

    if (contact!=null) {
      outState.putString("contact", contact.toString());
    }
  }

  private void restoreMe(Bundle state) {
```

```
    contact=null;

    if (state!=null) {
      String contactUri=state.getString("contact");

      if (contactUri!=null) {
        contact=Uri.parse(contactUri);
      }
    }
  }
}
```

By and large, it looks like a normal activity...because it is. Initially, the "model" – a `Uri` named `contact` – is `null`. It is set as the result of spawning the `ACTION_PICK` sub-activity. Its string representation is saved in `onSaveInstanceState()` and restored in `restoreMe()` (called from `onCreate()`). If the contact is not `null`, the "View" button is enabled and can be used to view the chosen contact.

Visually, it looks pretty much as one would expect:

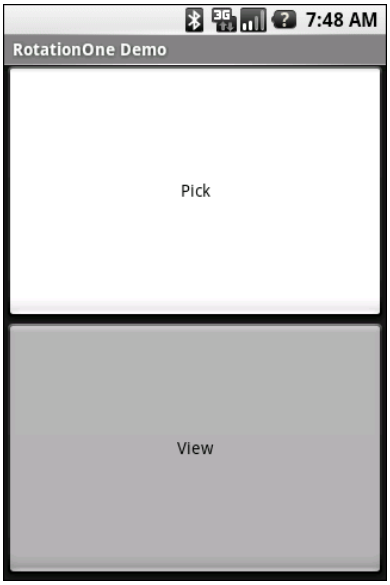


Figure 81. The RotationOne application, in portrait mode

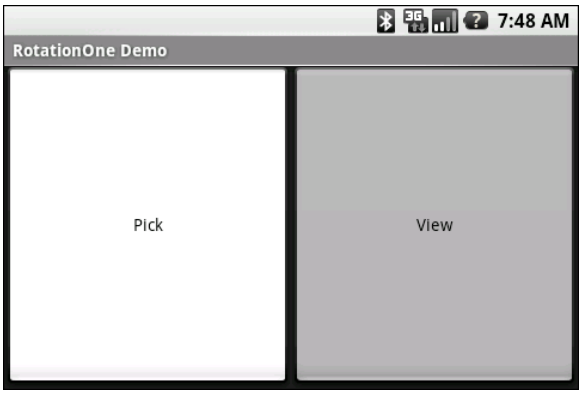


Figure 82. The RotationOne application, in landscape mode

The benefit to this implementation is that it handles a number of system events beyond mere rotation, such as being closed by Android due to low memory.

For fun, comment out the `restoreMe()` call in `onCreate()` and try running the application. You will see that the application "forgets" a contact selected in one orientation when you rotate the emulator or device.

Now With More Savings!

The problem with `onSaveInstanceState()` is that you are limited to a `Bundle`. That's because this callback is also used in cases where your whole process might be terminated (e.g., low memory), so the data to be saved has to be something that can be serialized and has no dependencies upon your running process.

For some activities, that limitation is not a problem. For others, though, it is more annoying. Take an online chat, for example. You have no means of storing a socket in a `Bundle`, so by default, you will have to drop your connection to the chat server and re-establish it. That not only may be a performance hit, but it might also affect the chat itself, such as you appearing in the chat logs as disconnecting and reconnecting.

One way to get past this is to use `onRetainNonConfigurationInstance()` instead of `onSaveInstanceState()` for "light" changes like a rotation. Your activity's `onRetainNonConfigurationInstance()` callback can return an `Object`, which you can retrieve later via `getLastNonConfigurationInstance()`. The `Object` can be just about anything you want – typically, it will be some kind of "context" object holding activity state, such as running threads, open sockets, and the like. Your activity's `onCreate()` can call `getLastNonConfigurationInstance()` – if you get a non-null response, you now have your sockets and threads and whatnot. The biggest limitation is that you do not want to put in the saved context anything that might reference a resource that will get swapped out, such as a `Drawable` loaded from a resource.

Let's take a look at the `Rotation/RotationTwo` sample project, which uses this approach to handling rotations. The layouts, and hence the visual appearance, is the same as with `Rotation/RotationOne`. Where things differ slightly is in the Java code:

```
public class RotationTwoDemo extends Activity {
  static final int PICK_REQUEST=1337;
  Button viewButton=null;
  Uri contact=null;

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

    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("content://contacts/people"));

        startActivityForResult(i, PICK_REQUEST);
      }
    });

    viewButton=(Button)findViewById(R.id.view);

    viewButton.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        startActivity(new Intent(Intent.ACTION_VIEW, contact));
      }
    });
```

```
    restoreMe();

    viewButton.setEnabled(contact!=null);
  }

  @Override
  protected void onActivityResult(int requestCode, int resultCode,
                                  Intent data) {
    if (requestCode==PICK_REQUEST) {
      if (resultCode==RESULT_OK) {
        contact=data.getData();
        viewButton.setEnabled(true);
      }
    }
  }

  @Override
  public Object onRetainNonConfigurationInstance() {
    return(contact);
  }

  private void restoreMe() {
    contact=null;

    if (getLastNonConfigurationInstance()!=null) {
      contact=(Uri)getLastNonConfigurationInstance();
    }
  }
}
```

In this case, we override `onRetainNonConfigurationInstance()`, returning the actual `Uri` for our contact, rather than a string representation of it. In turn, `restoreMe()` calls `getLastNonConfigurationInstance()`, and if it is not `null`, we hold onto it as our contact and enable the "View" button.

The advantage here is that we are passing around the `Uri` rather than a string representation. In this case, that is not a big savings. But our state could be much more complicated, including threads and sockets and other things we cannot pack into a `Bundle`.

DIY Rotation

Even this, though, may still be too intrusive to your application. Suppose, for example, you are creating a real-time game, such as a first-person shooter. The "hiccup" your users experience as your activity is destroyed and re-

created might be enough to get them shot, which they may not appreciate. While this would be less of an issue on the T-Mobile G1, since a rotation requires sliding open the keyboard and therefore is unlikely to be done mid-game, other devices might rotate based solely upon the device's position as determined by accelerometers.

The third possibility for handling rotations, therefore, is to tell Android that you will handle them completely yourself and that you do not want assistance from the framework. To do this:

1. Put an `android:configChanges` entry in your `AndroidManifest.xml` file, listing the configuration changes you want to handle yourself versus allowing Android to handle for you
2. Implement `onConfigurationChanged()` in your `Activity`, which will be called when one of the configuration changes you listed in `android:configChanges` occurs

Now, for any configuration change you want, you can bypass the whole activity-destruction process and simply get a callback letting you know of the change.

To see this in action, turn to the `Rotation/RotationThree` sample application. Once again, our layouts are the same, so the application looks the same as the preceding two samples. However, the Java code is significantly different, because we are no longer concerned with saving our state, but rather with updating our UI to deal with the layout.

But first, we need to make a small change to our manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.commonsware.android.rotation.three"
      android:versionCode="1"
      android:versionName="1.0.0">
    <application android:label="@string/app_name">
        <activity android:name=".RotationThreeDemo"
                  android:label="@string/app_name"
                  android:configChanges="keyboardHidden|orientation">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
```

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

Here, we state that we will handle `keyboardHidden` and `orientation` configuration changes ourselves. This covers us for any cause of the "rotation" – whether it is a sliding keyboard or a physical rotation. Note that this is set on the activity, not the application – if you have several activities, you will need to decide for each which of the tactics outlined in this chapter you wish to use.

The Java code for this project is shown below:

```
public class RotationThreeDemo extends Activity {
  static final int PICK_REQUEST=1337;
  Button viewButton=null;
  Uri contact=null;

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

    setupViews();
  }

  @Override
  protected void onActivityResult(int requestCode, int resultCode,
                                  Intent data) {
    if (requestCode==PICK_REQUEST) {
      if (resultCode==RESULT_OK) {
        contact=data.getData();
        viewButton.setEnabled(true);
      }
    }
  }

  public void onConfigurationChanged(Configuration newConfig) {
    super.onConfigurationChanged(newConfig);

    setupViews();
  }

  private void setupViews() {
    setContentView(R.layout.main);

    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("content://contacts/people"));

        startActivityForResult(i, PICK_REQUEST);
      }
    });

    viewButton=(Button)findViewById(R.id.view);

    viewButton.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        startActivity(new Intent(Intent.ACTION_VIEW, contact));
      }
    });

    viewButton.setEnabled(contact!=null);
  }
}
```

The `onCreate()` implementation delegates most of its logic to a `setupViews()` method, which loads the layout and sets up the buttons. The reason this logic was broken out into its own method is because it is also called from `onConfigurationChanged()`.

Forcing the Issue

In the previous three sections, we covered ways to deal with rotational events. There is, of course, a radical alternative: tell Android not to rotate your activity at all. If the activity does not rotate, you do not have to worry about writing code to deal with rotations.

To block Android from rotating your activity, all you need to do is add `android:screenOrientation = "portrait"` (or `"landscape"`, as you prefer) to your `AndroidManifest.xml` file, as shown below (from the `Rotation/RotationFour` sample project):

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.commonsware.android.rotation.four"
      android:versionCode="1"
      android:versionName="1.0.0">
    <application android:label="@string/app_name">
```

```
        <activity android:name=".RotationFourDemo"
                  android:screenOrientation="portrait"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Since this is applied on a per-activity basis, you will need to decide which of your activities may need this turned on.

At this point, your activity is locked into whatever orientation you specified, regardless of what you do. The following screen shots show the same activity as in the previous three sections, but using the above manifest and with the emulator set for both portrait and landscape orientation. Note that the UI does not move a bit, but remains in portrait mode.

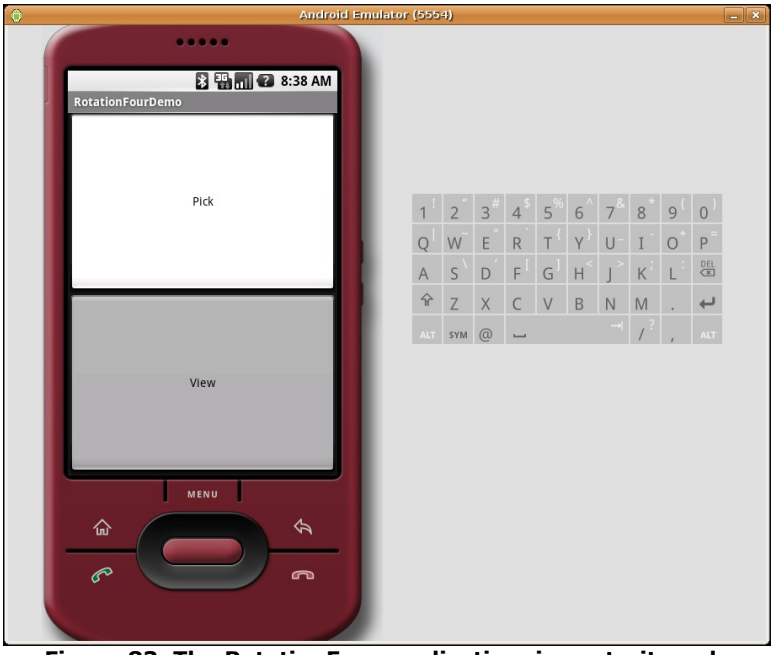


Figure 83. The RotationFour application, in portrait mode

Figure 84. The RotationFour application, in landscape mode

Making Sense of it All

All of these scenarios assume that you rotate the screen by opening up the keyboard on the device (or pressing `<Ctrl>-<F12>` in the emulator). Certainly, this is the norm for Android applications.

However, we haven't covered the iPhone Scenario.

You may have seen one (or several) commercials for the iPhone, showing how the screen rotates just by turning the device. By default, you do not get this behavior with the T-Mobile G1 – instead, the screen rotates based on whether the keyboard is open or closed.

However, it is very easy for you to change this behavior, so your screen will rotate based on the position of the phone: just add `android:screenOrientation = "sensor"` to your `AndroidManifest.xml` file (as seen in the `Rotation/RotationFive` sample project):

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.commonsware.android.rotation.five"
      android:versionCode="1"
      android:versionName="1.0.0">
    <application android:label="@string/app_name">
        <activity android:name=".RotationFiveDemo"
                  android:screenOrientation="sensor"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

The “sensor”, in this case, tells Android you want the accelerometers to control the screen orientation, so the physical shift in the device orientation controls the screen orientation.

At least on the G1, this appears to only work when going from the traditional upright portrait position to the traditional landscape position – rotating 90 degrees counter-clockwise. Rotating the device 90 degrees clockwise results in no change in the screen.

Also note that this setting disables having the keyboard trigger a rotation event. Leaving the device in the portrait position, if you slide out the keyboard, in a "normal" Android activity, the screen will rotate; in a `android:screenOrientation = "sensor"` activity, the screen will not rotate.

PART V – Content Providers and Services

CHAPTER 27

Using a Content Provider

Any `Uri` in Android that begins with the `content://` scheme represents a resource served up by a content provider. Content providers offer data encapsulation using `Uri` instances as handles – you neither know nor care where the data represented by the `Uri` comes from, so long as it is available to you when needed. The data could be stored in a SQLite database, or in flat files, or retrieved off a device, or be stored on some far-off server accessed over the Internet.

Given a `Uri`, you can perform basic CRUD (create, read, update, delete) operations using a content provider. `Uri` instances can represent either collections or individual pieces of content. Given a collection `Uri`, you can create new pieces of content via insert operations. Given an instance `Uri`, you can read data represented by the `Uri`, update that data, or delete the instance outright.

Android lets you use existing content providers, plus create your own. This chapter covers using content providers; the next chapter will explain how you can serve up your own data using the content provider framework.

Pieces of Me

The simplified model of the construction of a content `Uri` is the scheme, the namespace of data, and, optionally, the instance identifier, all separated by

CHAPTER 28

Building a Content Provider

Building a content provider is probably the most complicated and tedious task in all of Android development. There are many requirements of a content provider, in terms of methods to implement and public data members to supply. And, until you try using it, you have no great way of telling if you did any of it correctly (versus, say, building an activity and getting validation errors from the resource compiler).

That being said, building a content provider is of huge importance if your application wishes to make data available to other applications. If your application is keeping its data solely to itself, you may be able to avoid creating a content provider, just accessing the data directly from your activities. But, if you want your data to possibly be used by others – for example, you are building a feed reader and you want other programs to be able to access the feeds you are downloading and caching – then a content provider is right for you.

First, Some Dissection

As was discussed in the previous chapter, the content `Uri` is the linchpin behind accessing data inside a content provider. When using a content provider, all you really need to know is the provider's base `Uri`; from there you can run queries as needed, or construct a `Uri` to a specific instance if you know the instance identifier.

CHAPTER 29

Requesting and Requiring Permissions

In the late 1990's, a wave of viruses spread through the Internet, delivered via email, using contact information culled from Microsoft Outlook. A virus would simply email copies of itself to each of the Outlook contacts that had an email address. This was possible because, at the time, Outlook did not take any steps to protect data from programs using the Outlook API, since that API was designed for ordinary developers, not virus authors.

Nowadays, many applications that hold onto contact data secure that data by requiring that a user explicitly grant rights for other programs to access the contact information. Those rights could be granted on a case-by-case basis or a once at install time.

Android is no different, in that it requires permissions for applications to read or write contact data. Android's permission system is useful well beyond contact data, and for content providers and services beyond those supplied by the Android framework.

You, as an Android developer, will frequently need to ensure your applications have the appropriate permissions to do what you want to do with other applications' data. You may also elect to require permissions for other applications to use your data or services, if you make those available to other Android components. This chapter covers how to accomplish both these ends.

CHAPTER 30

Creating a Service

As noted previously, Android services are for long-running processes that may need to keep running even when decoupled from any activity. Examples include playing music even if the "player" activity gets garbage-collected, polling the Internet for RSS/Atom feed updates, and maintaining an online chat connection even if the chat client loses focus due to an incoming phone call.

Services are created when manually started (via an API call) or when some activity tries connecting to the service via inter-process communication (IPC). Services will live until no longer needed and if RAM needs to be reclaimed. Running for a long time isn't without its costs, though, so services need to be careful not to use too much CPU or keep radios active too much of the time, lest the service cause the device's battery to get used up too quickly.

This chapter covers how you can create your own services; the next chapter covers how you can use such services from your activities or other contexts. Both chapters will analyze the `Service/WeatherPlus` sample application, with this chapter focusing mostly on the `WeatherPlusService` implementation. `WeatherPlusService` extends the weather-fetching logic of the original `Internet/Weather` sample, by bundling it in a service that monitors changes in location, so the weather is updated as the emulator is "moved".

CHAPTER 31

Invoking a Service

Services can be used by any application component that "hangs around" for a reasonable period of time. This includes activities, content providers, and other services. Notably, it does not include pure intent receivers (i.e., intent receivers that are not part of an activity), since those will get garbage collected immediately after each instance processes one incoming `Intent`.

To use a service, you need to get an instance of the AIDL interface for the service, then call methods on that interface as if it were a local object. When done, you can release the interface, indicating you no longer need the service.

In this chapter, we will look at the client side of the `Service/WeatherPlus` sample application. The `WeatherPlus` activity looks an awful lot like the original `Weather` application – just a Web page showing a weather forecast:

CHAPTER 32

Alerting Users Via Notifications

Pop-up messages. Tray icons and their associated "bubble" messages. Bouncing dock icons. You are no doubt used to programs trying to get your attention, sometimes for good reason.

Your phone also probably chirps at you for more than just incoming calls: low battery, alarm clocks, appointment notifications, incoming text message or email, etc.

Not surprisingly, Android has a whole framework for dealing with these sorts of things, collectively called "notifications".

Types of Pestering

A service, running in the background, needs a way to let users know something of interest has occurred, such as when email has been received. Moreover, the service may need some way to steer the user to an activity where they can act upon the event – reading a received message, for example. For this, Android supplies status bar icons, flashing lights, and other indicators collectively known as "notifications".

Your current phone may well have such icons, to indicate battery life, signal strength, whether Bluetooth is enabled, and the like. With Android, applications can add their own status bar icons, with an eye towards having them appear only when needed (e.g., a message has arrived).

PART VI – Other Android Capabilities and Tools

CHAPTER 33

Accessing Location-Based Services

A popular feature on current-era mobile devices is GPS capability, so the device can tell you where you are at any point in time. While the most popular use of GPS service is mapping and directions, there are other things you can do if you know your location. For example, you might set up a dynamic chat application where the people you can chat with are based on physical location, so you're chatting with those you are nearest. Or, you could automatically "geotag" posts to Twitter or similar services.

GPS is not the only way a mobile device can identify your location. Alternatives include:

- The European equivalent to GPS, called Galileo, which is still under development at the time of this writing
- Cell tower triangulation, where your position is determined based on signal strength to nearby cell towers
- Proximity to public WiFi "hotspots" that have known geographic locations

Android devices may have one or more of these services available to them. You, as a developer, can ask the device for your location, plus details on what providers are available. There are even ways for you to simulate your location in the emulator, for use in testing your location-enabled applications.

CHAPTER 34

Mapping with MapView and MapActivity

One of Google's most popular services – after search, of course – is Google Maps, where you can find everything from the nearest pizza parlor to directions from New York City to San Francisco (only 2,905 miles!) to street views and satellite imagery.

Android, not surprisingly, integrates Google Maps. There is a mapping activity available to users straight off the main Android launcher. More relevant to you, as a developer, are `MapView` and `MapActivity`, which allow you to integrate maps into your own applications. Not only can you display maps, control the zoom level, and allow people to pan around, but you can tie in Android's location-based services to show where the device is and where it is going.

Fortunately, integrating basic mapping features into your Android project is fairly easy. However, there is a fair bit of power available to you, if you want to get fancy.

Terms, Not of Endearment

Google Maps, particularly when integrated into third party applications, requires agreeing to a fairly lengthy set of legal terms. These terms include clauses that you may find unpalatable.

CHAPTER 35

Handling Telephone Calls

Many, if not most, Android devices will be phones. As such, not only will users be expecting to place and receive calls using Android, but you will have the opportunity to help them place calls, if you wish.

Why might you want to?

- Maybe you are writing an Android interface to a sales management application (a la Salesforce.com) and you want to offer users the ability to call prospects with a single button click, and without them having to keep those contacts both in your application and in the phone's contacts application
- Maybe you are writing a social networking application, and the roster of phone numbers that you can access shifts constantly, so rather than try to "sync" the social network contacts with the phone's contact database, you let people place calls directly from your application
- Maybe you are creating an alternative interface to the existing contacts system, perhaps for users with reduced motor control (e.g., the elderly), sporting big buttons and the like to make it easier for them to place calls

Whatever the reason, Android has the means to let you manipulate the phone just like any other piece of the Android system.

CHAPTER 36

Searching with SearchManager

One of the firms behind the Open Handset Alliance – Google – has a teeny weeny Web search service, one you might have heard of in passing. Given that, it's not surprising that Android has some amount of built-in search capabilities.

Specifically, Android has "baked in" the notion of searching not only on the device for data, but over the air to Internet sources of data.

Your applications can participate in the search process, by triggering searches or perhaps by allowing your application's data to be searched.

Note that this is fairly new to the Android platform, and so some shifting in the APIs is likely. Stay tuned for updates to this chapter.

Hunting Season

There are two types of search in Android: local and global. Local search searches within the current application; global search searches the Web via Google's search engine. You can initiate either type of search in a variety of ways, including:

- You can call `onSearchRequested()` from a button or menu choice, which will initiate a local search (unless you override this method in your activity)

CHAPTER 37

Development Tools

The Android SDK is more than a library of Java classes and API calls. It also includes a number of tools to assist in application development.

Much of the focus has been on the Eclipse plug-in, to integrate Android development with that IDE. Secondary emphasis has been placed on the plug-in's equivalents for use in other IDEs or without an IDE, such as `adb` for communicating with a running emulator.

This chapter will cover other tools beyond those two groups.

Hierarchical Management

Android comes with a Hierarchy Viewer tool, designed to help you visualize your layouts as they are seen in a running activity in a running emulator. So, for example, you can determine how much space a certain widget is taking up, or try to find where a widget is hiding that does not appear on the screen.

To use the Hierarchy Viewer, you first need to fire up your emulator, install your application, launch your activity, and navigate to spot you wish to examine. For illustration purposes, we'll use the ReadWrite demo application we introduced back in the chapter on file access:

Figure 97. ReadWrite demo application

You can launch the Hierarchy Viewer via the `hierarchyviewer` program, found in the `tools/` directory in your Android SDK installation. This brings up the main Hierarchy Viewer window:

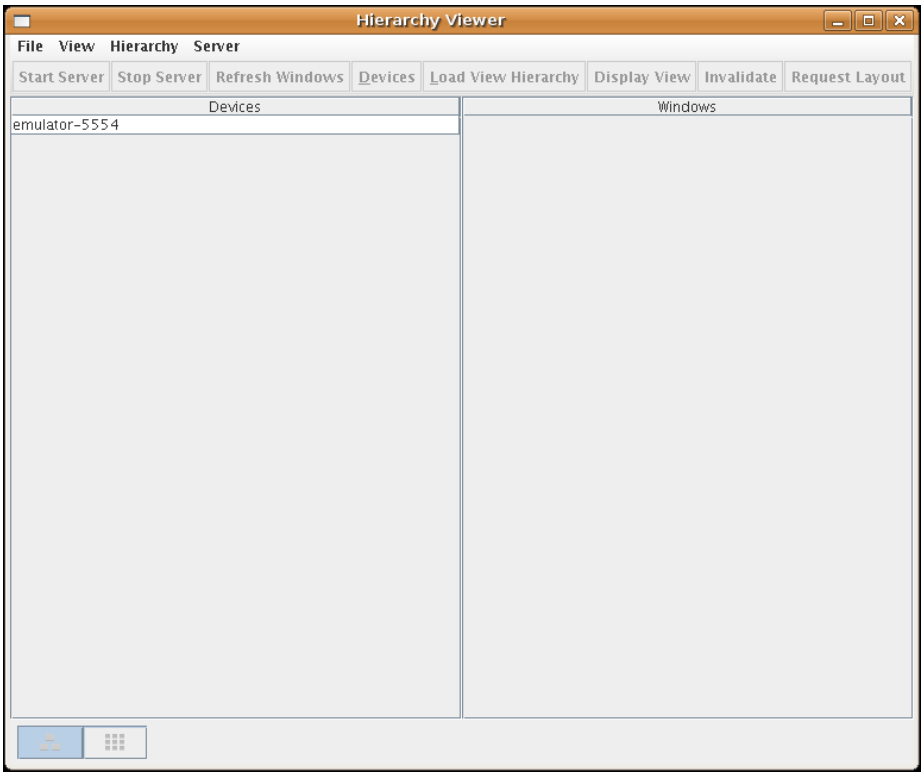


Figure 98. Hierarchy Viewer main window

The list on the left shows the various emulators you have opened. The number after the hyphen should line up with the number in parentheses in your emulator's title bar.

Clicking on an emulator shows, on the right, the list of "windows" available for examination:

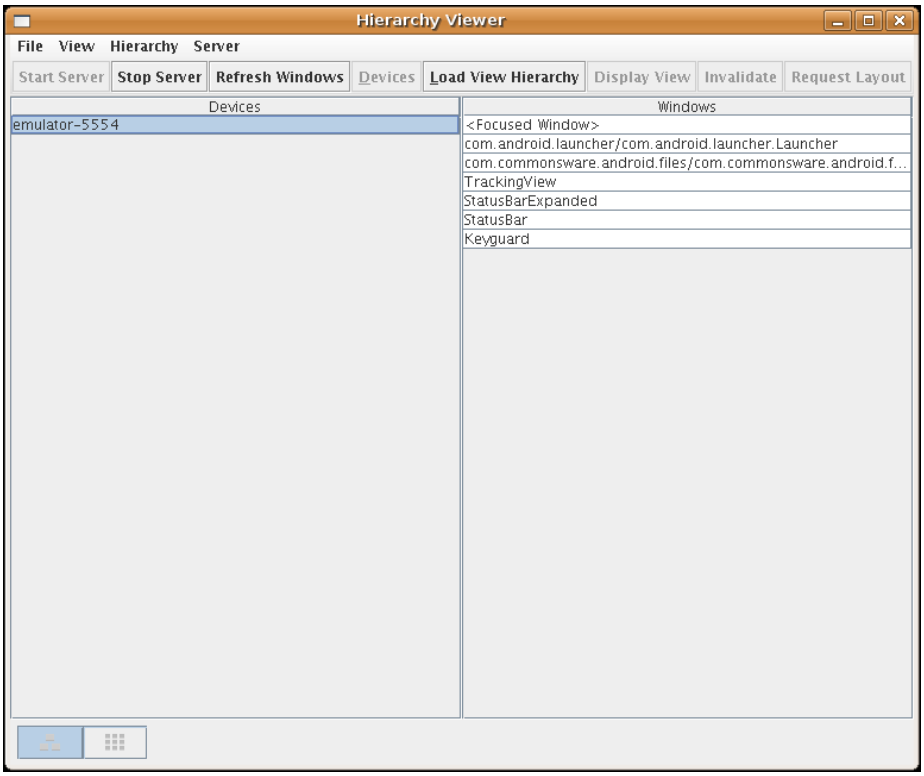


Figure 99. Hierarchy Viewer list of available windows

Note how there are many other windows besides our open activity, including the Launcher (i.e., the home screen), the Keyguard (i.e., the "Press Menu to Unlock" black screen you get when first opening the emulator), and so on. Your activity will be identified by application package and class (e.g., `com.commonsware.android.files/...`).

Where things get interesting, though, is when you choose a window and click Load View Hierarchy. After a few seconds, the details spring into view, in a perspective called the Layout View:

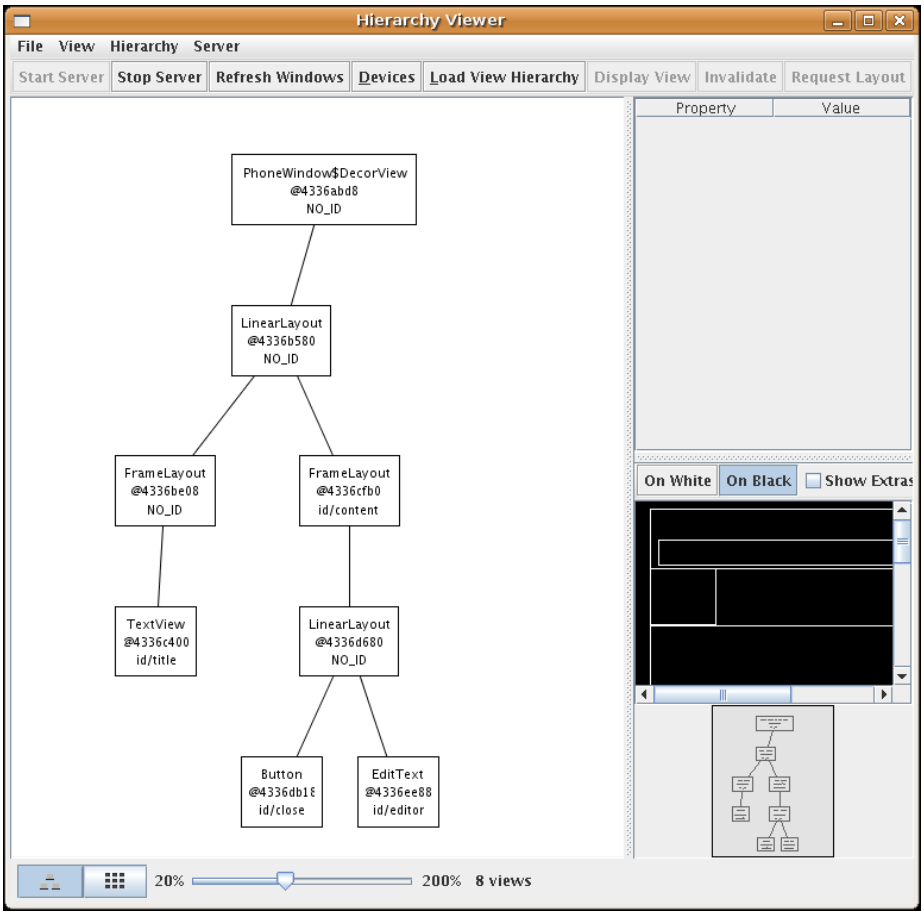


Figure 100. Hierarchy Viewer Layout View

The main area of the Layout View shows a tree of the various Views that make up your activity, starting from the overall system window and driving down into the individual UI widgets that users are supposed to interact with. You will see, on the lower-right branch of the tree, the `LinearLayout`, `Button`, and `EditText` shown in the above code listing. The remaining `Views` are all supplied by the system, including the title bar.

Clicking on one of the views adds more information to this perspective:

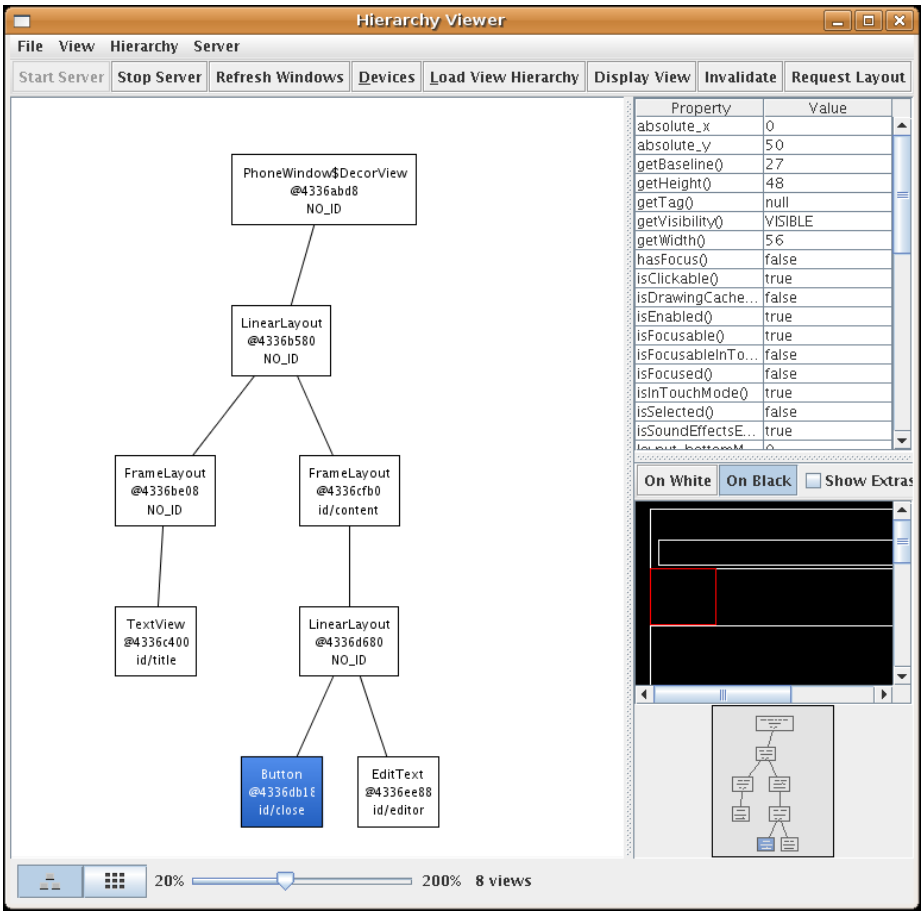


Figure 101. Hierarchy Viewer View properties

Now, in the upper-right region of the Viewer, we see properties of the selected widget — in this case, the `Button`. Alas, these properties do not appear to be editable.

Also, the widget is highlighted in red in the wireframe of the activity, shown beneath the properties (by default, views are shown as white outlines on a black background). This can help you ensure you have selected the right widget, if, say, you have several buttons and cannot readily tell from the tree what is what.

And, if you double-click on a `View` in the tree, you are given a pop-up pane showing just that `View` (and its children), isolated from the rest of your activity.

Down in the lower-left corner, you will see two toggle buttons, with the tree button initially selected. Clicking on the grid button changes puts the Viewer in a whole new perspective, called the Pixel Perfect View:

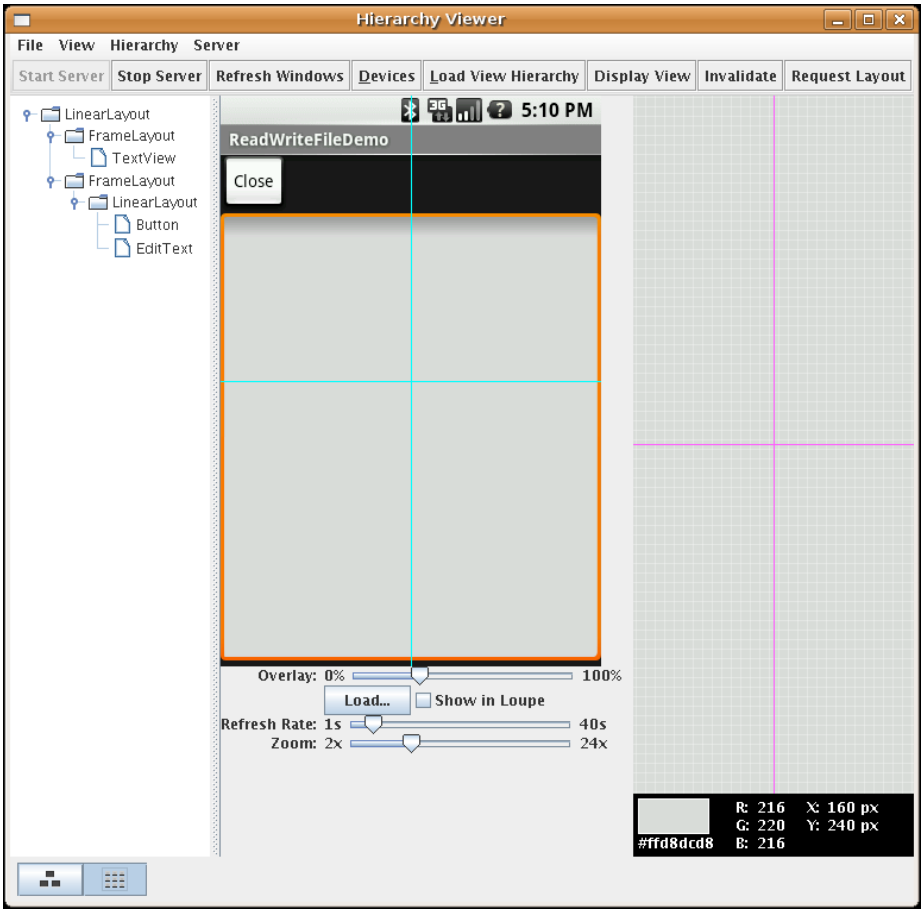


Figure 102. Hierarchy Viewer Pixel Perfect View

On the left, you see a tree representing the widgets and other `Views` in your activity. In the middle, you see your activity (the Normal View), and on the right, you see a zoomed edition of your activity (the Loupe View).

What may not be initially obvious is that this imagery is live. Your activity is polled every so often, controlled by the Refresh Rate slider. Anything you do in the activity will then be reflected in the Pixel Perfect View's Normal and Loupe Views.

The hairlines (cyan) overlaying the activity show the position being zoomed upon — just click on a new area to change where the Loupe View is inspecting. And, of course, there is another slider to adjust how much the Loupe View is zoomed.

Delightful Dalvik Debugging Detailed, De-moed

Another tool in the Android developer's arsenal is the Dalvik Debug Monitor Service (DDMS). This is a "Swiss army knife", allowing you to do everything from browse log files, update the GPS location provided by emulator, simulate incoming calls and messages, and browse the on-emulator storage to push and pull files.

Eventually, this section will contain a complete overview of DDMS. However, DDMS has a wide range of uses, so this section will gradually expand over time to try to cover them all.

To launch DDMS, run the `ddms` program inside the `tools/` directory in your Android SDK distribution. It will initially display just a tree of emulators and running programs on the left:

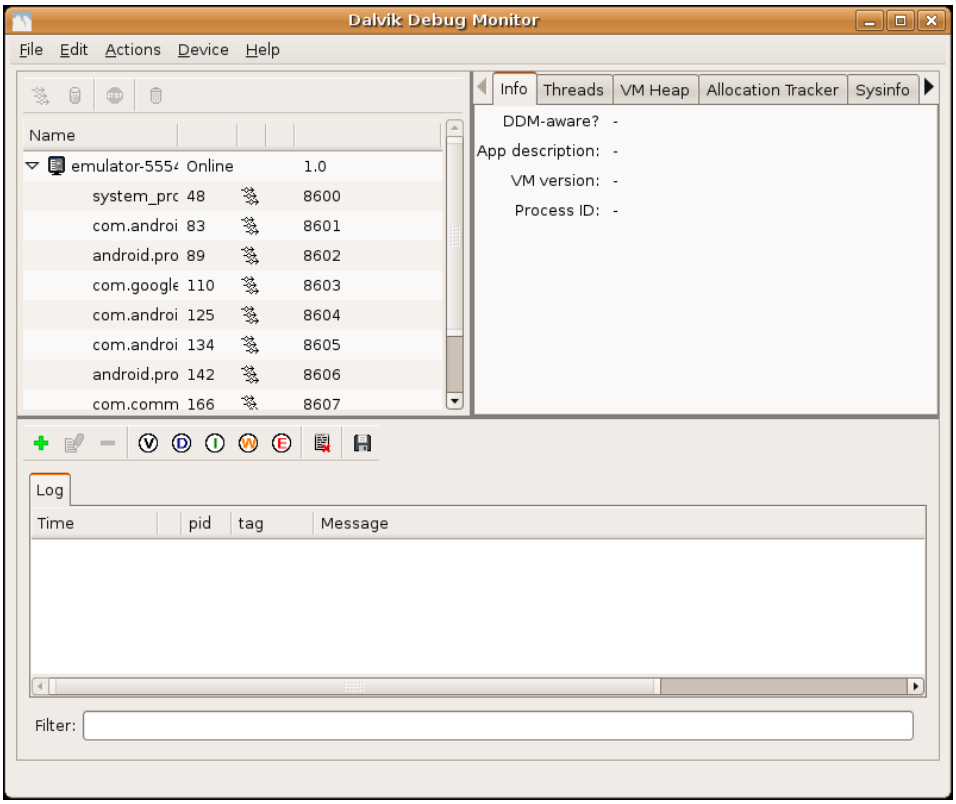


Figure 103. DDMS initial view

Clicking on an emulator allows you to browse the event log on the bottom and manipulate the emulator via the tabs on the right:

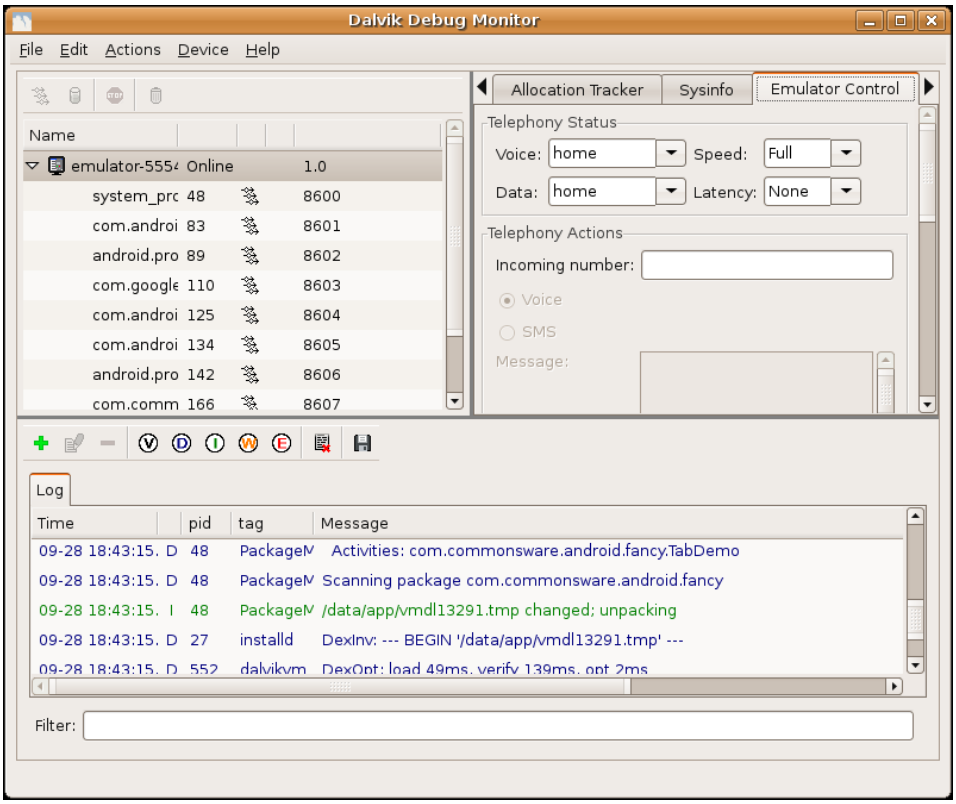


Figure 104. DDMS, with emulator selected

Logging

Rather than use `adb logcat`, DDMS lets you view your logging information in a scrollable table. Just highlight the emulator or device you want to monitor, and the bottom half of the screen shows the logs.

In addition, you can:

- Filter the Log tab by any of the five logging levels, shown as the V through E toolbar buttons.
- Create a custom filter, so you can view only those tagged with your application's tag, by pressing the + toolbar button and completing the form (shown below). The name you enter in the form will be

used as the name of another logging output tab in the bottom portion of the DDMS main window.

- Save the log information to a text file for later perusal, or for searching.

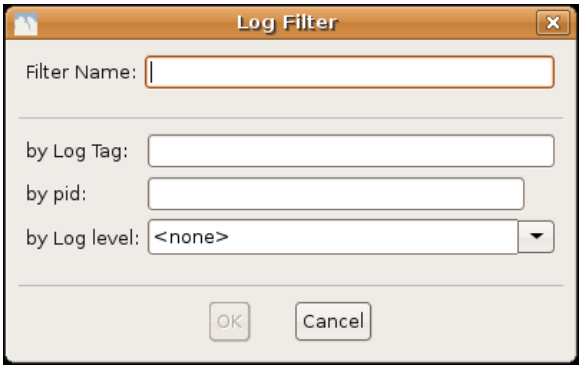


Figure 105. DDMS logging filter

File Push and Pull

While you can use `adb pull` and `adb push` to get files to and from an emulator or device, DDMS lets you do that visually. Just highlight the emulator or device you wish to work with, then choose **Device|File Explorer...** from the main menu. That will bring up your typical directory browser:

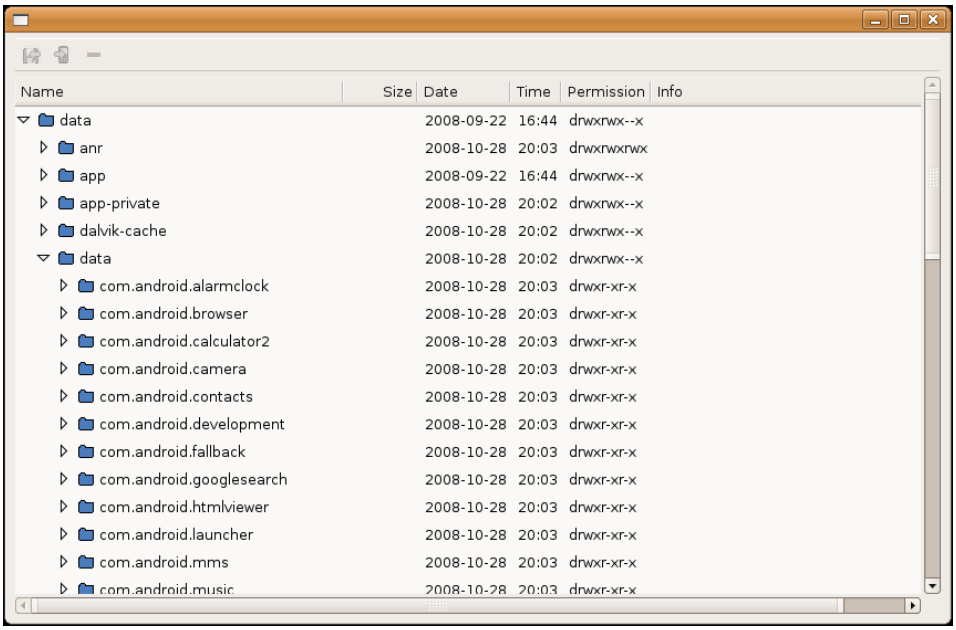


Figure 106. DDMS File Explorer

Just browse to the file you want and click either the pull (left-most) or push (middle) toolbar button to transfer the file to/from your development machine. Or, click the delete (right-most) toolbar button to delete the file.

There are a few caveats to this:

- You cannot create directories through this tool. You will either need to use `adb shell` or create them from within your application.
- While you can putter through most of the files on an emulator, you can access very little outside of `/sdcard` on an actual device, due to Android security restrictions.

Screenshots

To take a screenshot of the Android emulator or device, simply press `<Ctrl>-<S>` or choose **Device| Screen capture...** from the main menu. This will bring up a dialog box containing an image of the current screen:

Figure 107. DDMS screen capture

From here, you can click **[Save]** to save the image as a PNG file somewhere on your development machine, **[Refresh]** to update the image based on the current state of the emulator or device, or **[Done]** to close the dialog.

Location Updates

To use DDMS to supply location updates to your application, the first thing you must do is have your application use the `gps` `LocationProvider`, as that is the one that DDMS is set to update.

Then, click on the Emulator Control tab and scroll down to the Location Controls section. Here, you will find a smaller tabbed pane with three options for specifying locations: Manual, GPX, and KML:

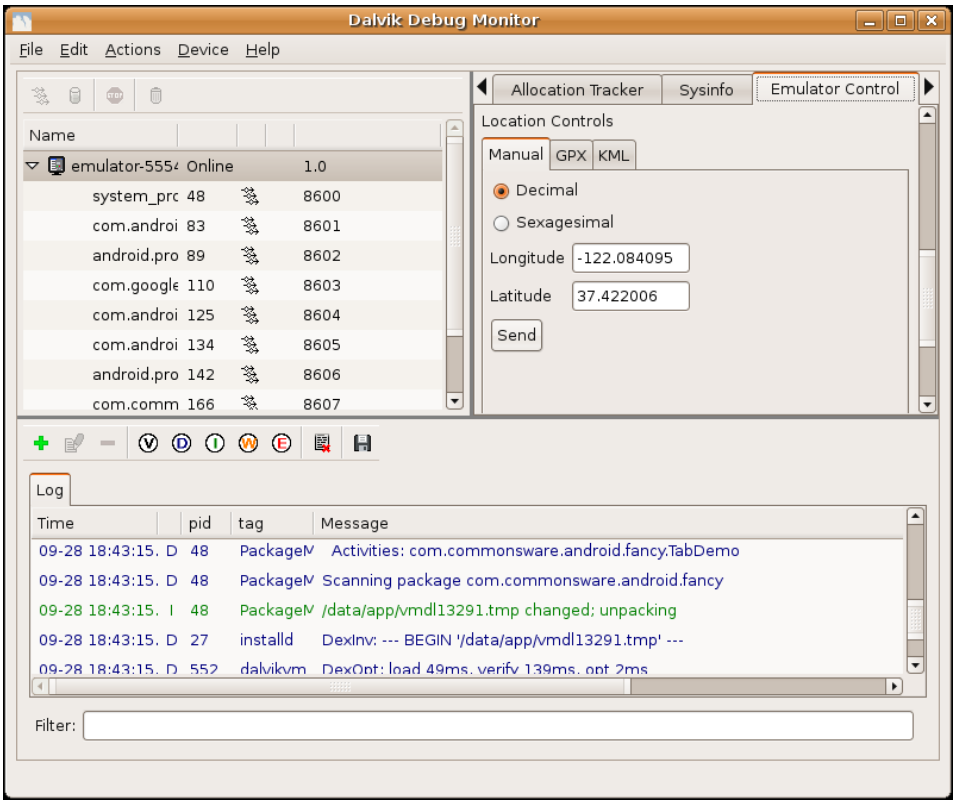


Figure 108. DDMS location controls

The Manual tab is fairly self-explanatory: provide a latitude and longitude and click the Send button to submit that location to the emulator. The emulator, in turn will notify any location listeners of the new position.

Discussion of the GPX and KML options is reserved for a future edition of this book.

Placing Calls and Messages

If you want to simulate incoming calls or SMS messages to the Android emulator, DDMS can handle that as well.

On the Emulator Control tab, above the Location Controls group, is the Telephony Actions group:

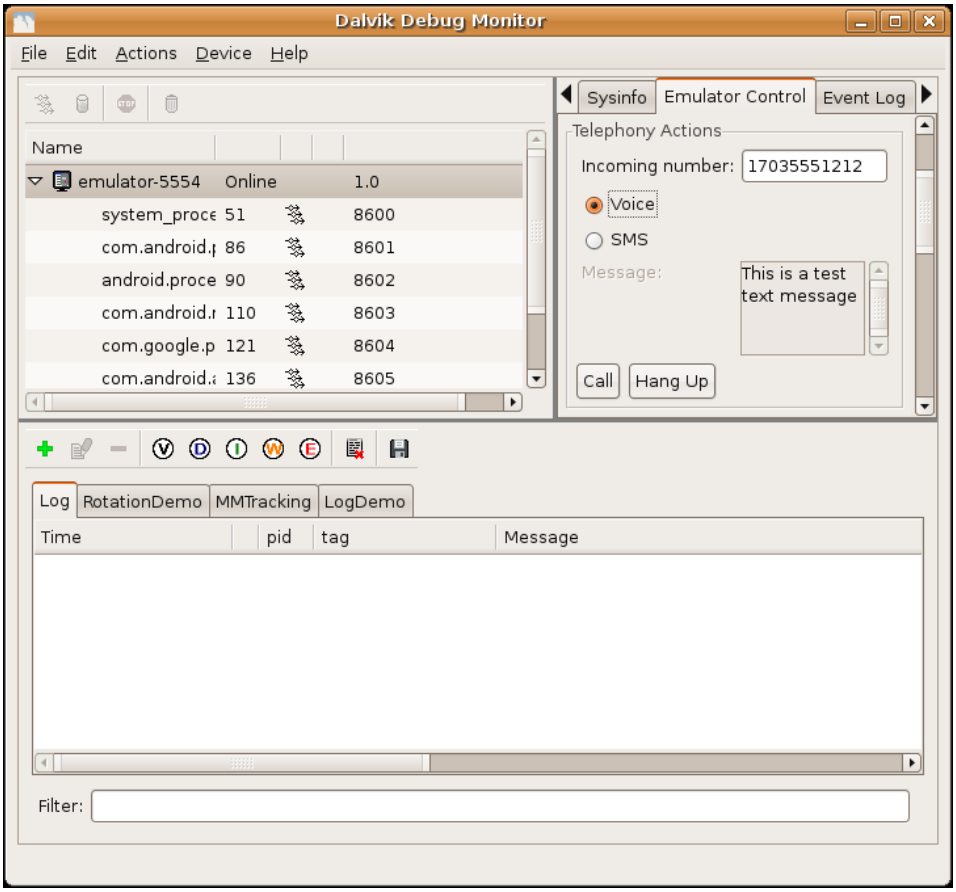


Figure 109. DDMS telephony controls

To simulate an incoming call, fill in a phone number, choose the Voice radio button, and click Call. At that point, the emulator will show the incoming call, allowing you to accept it (via the green phone button) or reject it (via the red phone button):

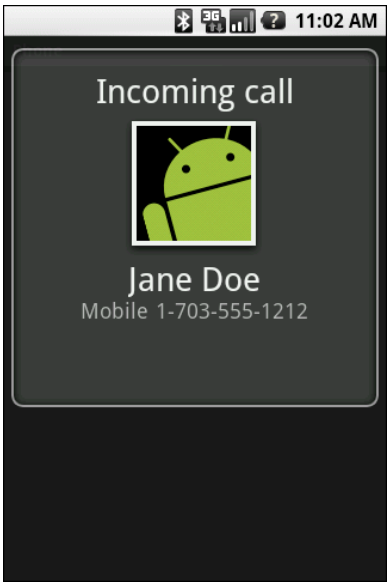


Figure 110. Simulated incoming call

To simulate in an incoming text message, fill in a phone number, choose the SMS radio button, enter a message in the provided text area, and click Send. The text message will then appear as a notification:

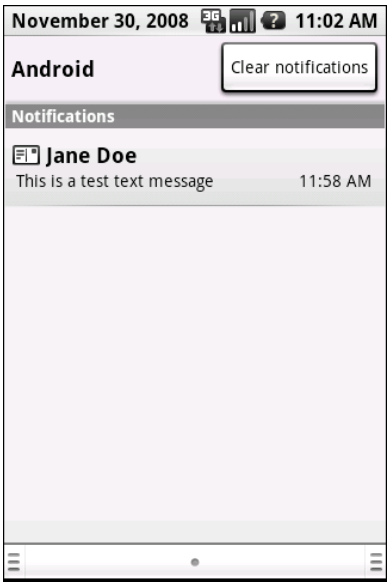


Figure 111. Simulated text message

And, of course, you can click on the notification to view the message in the full-fledged Messaging application:

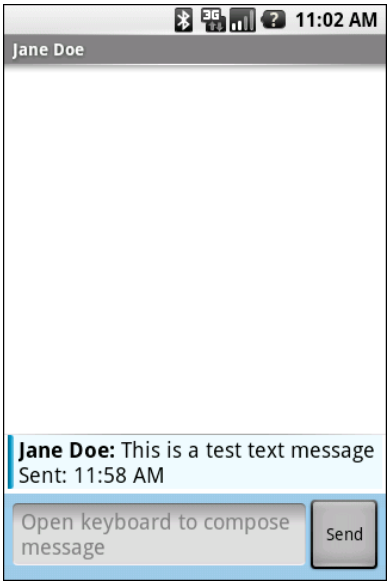


Figure 112. Simulated text message, in Messaging application

Put It On My Card

The T-Mobile G1 has a microSD card slot. Many other Android devices are likely to have similar forms of removable storage, which the Android platform refers to generically as an "SD card".

SD cards are strongly recommended to be used by developers as the holding pen for large data sets: images, movie clips, audio files, etc. The T-Mobile G1, in particular, has a relatively paltry amount of on-board flash memory, so the more you can store on an SD card, the better.

Of course, the challenge is that, while the G1 has an SD card by default, the emulator does not. To make the emulator work like the G1, you need to create and "insert" an SD card into the emulator.

Creating a Card Image

Rather than require emulators to somehow have access to an actual SD card reader and use actual SD cards, Android is set up to use card images. An image is simply a file that the emulator will treat as if it were an SD card volume. If you are used to disk images used with virtualization tools (e.g., VirtualBox), the concept is the same: Android uses a disk image representing the SD card contents.

To create such an image, use the `mksdcard` utility, provided in the `tools/` directory of your SDK installation. This takes two main parameters:

1. The size of the image, and hence the size of the resulting "card". If you just supply a number, it is interpreted as a size in bytes. Alternatively, you can append `K` or `M` to the number to indicate a size in kilobytes or megabytes, respectively.
2. The filename under which to store the image.

So, for example, to create a 1GB SD card image, to simulate the G1's SD card in the emulator, you could run:

```
mksdcard 1024M sdcard.img
```

"Inserting" the Card

To have your emulator use this SD card image, start the emulator with the `-sdcard` switch, containing a fully-qualified path to the image file you created using `mksdcard`. While there will be no visible impact – there is no icon or anything in Android showing that you have a card mounted – the `/sdcard` path will now be available for reading and writing.

To put files on the `/sdcard`, either use the File Explorer in DDMS or `adb push` and `adb pull` from the console.

CHAPTER 38

Where Do We Go From Here?

Obviously, this book does not cover everything. And while your #1 resource (besides the book) is going to be the Android SDK documentation, you are likely to need information beyond what's covered in either of those places.

Searching online for "android" and a class name is a good way to turn up tutorials that reference a given Android class. However, bear in mind that tutorials written before late August 2008 are probably written for the M5 SDK and, as such, will require considerable adjustment to work properly in current SDKs.

Beyond randomly hunting around for tutorials, though, this chapter outlines some other resources to keep in mind.

Questions. Sometimes, With Answers.

The "official" places to get assistance with Android are the Android Google Groups. With respect to the SDK, there are three to consider following:

- android-beginners, a great place to ask entry-level questions
- android-developers, best suited for more complicated questions or ones that delve into less-used portions of the SDK
- android-discuss, designed for free-form discussion of anything Android-related, not necessarily for programming questions and answers

Keyword Index

Class.....

Command.....

Constant.....

Method....

Property....