

Utilize the Android media APIs
to create dynamic mobile apps



Pro Android Media

Developing Graphics, Music, Video and
Rich Media Apps for Smartphones and Tablets

Shawn Van Every

Apress®

Pro Android Media

Developing Graphics, Music, Video,
and Rich Media Apps for Smartphones
and Tablets



Shawn Van Every

Apress®

Pro Android Media: Developing Graphics, Music, Video, and Rich Media Apps for Smartphones and Tablets

Copyright © 2009 by Shawn Van Every

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN 978-1-4302-3267-4

ISBN 978-1-4302-3268-1 (eBook)

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

President and Publisher: Paul Manning

Lead Editor: Matthew Moodie

Technical Reviewers: Steve Bull and Wallace Jackson

Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Coordinating Editor: Corbin Collins

Copy Editor: Mary Ann Fugate

Compositor: MacPS, LLC

Indexer: BIM Indexing & Proofreading Services

Artist: April Milne

Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media, LLC., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at www.apress.com/info/bulksales.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at www.apress.com.

Contents at a Glance

■ Contents	iv
■ About the Author	viii
■ About the Technical Reviewers	ix
■ Acknowledgments	x
■ Preface	xi
■ Chapter 1: Introduction to Android Imaging.....	1
■ Chapter 2: Building Custom Camera Applications.....	23
■ Chapter 3: Image Editing and Processing	47
■ Chapter 4: Graphics and Touch Events.....	79
■ Chapter 5: Introduction to Audio on Android.....	105
■ Chapter 6: Background and Networked Audio	125
■ Chapter 7: Audio Capture	151
■ Chapter 8: Audio Synthesis and Analysis.....	179
■ Chapter 9: Introduction to Video	195
■ Chapter 10: Advanced Video	211
■ Chapter 11: Video Capture.....	229
■ Chapter 12: Media Consumption and Publishing Using Web Services	251
■ Index.....	291

Contents

■ Contents at a Glance	iii
■ About the Author	viii
■ About the Technical Reviewers	ix
■ Acknowledgments	x
■ Preface	xi
■ Chapter 1: Introduction to Android Imaging	1
Image Capture Using the Built-In Camera Application.....	1
Returning Data from the Camera App.....	3
Capturing Larger Images	5
Displaying Large Images.....	6
Image Storage and Metadata.....	10
Obtaining an URI for the Image.....	11
Updating Our CameraActivity to Use MediaStore for Image Storage and to Associate Metadata	12
Retrieving Images Using the MediaStore.....	16
Creating an Image Viewing Application	18
Internal Metadata.....	21
Summary	21
■ Chapter 2: Building Custom Camera Applications	23
Using the Camera Class.....	23
Camera Permissions	24
Preview Surface.....	24
Implementing the Camera	25
Putting It All Together	35
Extending the Custom Camera Application.....	38
Building a Timer-Based Camera App.....	38
Building a Time-Lapse Photography App.....	43
Summary	45

■ Chapter 3: Image Editing and Processing	47
Selecting Images Using the Built-In Gallery Application	47
Drawing a Bitmap onto a Bitmap	52
Basic Image Scaling and Rotating	54
Enter the Matrix	55
Matrix Methods	58
Alternative to Drawing	64
Image Processing	65
ColorMatrix	65
Altering Contrast and Brightness	67
Changing Saturation	69
Image Compositing	69
Summary	78
■ Chapter 4: Graphics and Touch Events.....	79
Canvas Drawing	79
Bitmap Creation	79
Bitmap Configuration	80
Creating the Canvas.....	81
Working with Paint.....	82
Drawing Shapes.....	83
Drawing Text.....	87
Finger Painting.....	93
Touch Events	93
Drawing on Existing Images	97
Saving a Bitmap-Based Canvas Drawing.....	101
Summary	104
■ Chapter 5: Introduction to Audio on Android.....	105
Audio Playback	105
Supported Audio Formats	106
Using the Built-In Audio Player via an Intent	107
Creating a Custom Audio-Playing Application	109
MediaStore for Audio	115
Summary	123
■ Chapter 6: Background and Networked Audio	125
Background Audio Playback	125
Services	125
Local Service plus MediaPlayer	129
Controlling a MediaPlayer in a Service	132
Networked Audio	137
HTTP Audio Playback	137
Streaming Audio via HTTP	143
RTSP Audio Streaming	150
Summary	150
■ Chapter 7: Audio Capture	151
Audio Capture with an Intent	151
Custom Audio Capture	154

MediaRecorder Audio Sources.....	155
MediaRecorder Output Formats.....	155
MediaRecorder Audio Encoders.....	156
MediaRecorder Output and Recording.....	156
MediaRecorder State Machine.....	156
MediaRecorder Example.....	157
Other MediaRecorder Methods.....	162
Inserting Audio into the MediaStore.....	167
Raw Audio Recording with AudioRecord.....	167
Raw Audio Playback with AudioTrack.....	170
Raw Audio Capture and Playback Example.....	172
Summary.....	177
Chapter 8: Audio Synthesis and Analysis.....	179
Digital Audio Synthesis.....	179
Playing a Synthesized Sound.....	180
Generating Samples.....	182
Audio Analysis.....	187
Capturing Sound for Analysis.....	188
Visualizing Frequencies.....	189
Summary.....	193
Chapter 9: Introduction to Video.....	195
Video Playback.....	195
Supported Formats.....	195
Playback Using an Intent.....	196
Playback Using VideoView.....	197
Adding Controls with MediaController.....	199
Playback Using a MediaPlayer.....	200
Summary.....	210
Chapter 10: Advanced Video.....	211
MediaStore for Retrieving Video.....	211
Video Thumbnails from the MediaStore.....	212
Full MediaStore Video Example.....	212
Networked Video.....	218
Supported Network Video Types.....	218
Network Video Playback.....	221
Summary.....	228
Chapter 11: Video Capture.....	229
Recording Video Using an Intent.....	229
Adding Video Metadata.....	232
Custom Video Capture.....	235
MediaRecorder for Video.....	235
Full Custom Video Capture Example.....	246
Summary.....	250
Chapter 12: Media Consumption and Publishing Using Web Services.....	251
Web Services.....	251
HTTP Requests.....	252

JSON	254
Pulling Flickr Images Using JSON	257
Location	263
Pulling Flickr Images Using JSON and Location	266
REST	273
Representing Data in XML	273
SAX Parsing	274
HTTP File Uploads	278
Making an HTTP Request	278
Uploading Video to Blip.TV	280
Summary	290
■ Index	291

About the Author



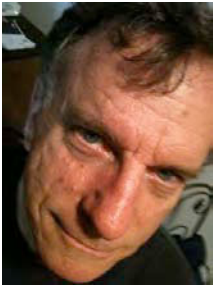
Shawn Van Every runs a mobile and streaming media consultancy to help companies better utilize emerging technologies related to audio and video with a focus on mobile and streaming applications. His clients have ranged from 19 Entertainment, MoMA, and Disney to Morgan Stanley, Lehman Brothers, and NYU Medical School, along with countless start-ups and other small clients.

Additionally, Shawn is an Adjunct Assistant Professor of Communication in NYU's Interactive Telecommunications Program. His teaching is varied and includes courses on participatory and social media, programming, mobile technologies, and interactive telephony. In 2008 he was honored with the David Payne Carter award for excellence in teaching.

He has demonstrated, exhibited, and presented work at many conferences and technology demonstrations, including O'Reilly's Emerging Telephony, O'Reilly's Emerging Technology, ACM Multimedia, Vloggercon, and Strong Angel II. He was a co-organizer of the Open Media Developers Summit, Beyond Broadcast (2006), and iPhoneDevCamp NYC.

Shawn holds a Master's degree in Interactive Telecommunications from NYU and a Bachelor's degree in Media Study from SUNY at Buffalo.

About the Technical Reviewers



Steve Bull has been coding and manipulating mobile devices since his days at Paul Allen's Interval Research in Palo Alto. As a mixed-media technology artist and entrepreneur, for the last nine years Bull has created location-specific narratives and games that explore the social, technological and creative possibilities of cell phones. He can be reached at www.stevebull.org.

Wallace Jackson is a seasoned multimedia producer and i3D programmer for Acrobat3D PDF, Android mobile apps, iTV Design, JavaFX, and JavaTV. He has been designing rich media since the Atari ST1040 and AMIGA 3000 and has been writing for leading multimedia publications on new media content development since the advent of *Multimedia Producer* magazine nearly two decades ago. He can be reached at www.wallacejackson.com.

Acknowledgments

The idea for this book came out of my work teaching at NYU. A huge debt of gratitude is owed to the ever encouraging faculty, staff, and students who comprise NYU's Interactive Telecommunications Program and who provide an endless source of inspiration. Thank you to Red Burns for creating, fostering, and ever improving ITP. Thank you to Dan O'Sullivan for constantly challenging me. Thank you to Tom Igoe and Dan Shiffman for showing me that it can be done. Thank you to Rob Ryan and Marianne Petite for all of your support. Thank you to all of the rest of the faculty, staff, and residents that I have worked with. And thank you to all of my current and former students who have made me realize how rewarding it can be to teach and see projects come alive; particularly Nisma Zaman, who provided very valuable early feedback.

This book would not have come close to being in existence if it weren't for the dedicated and very talented staff at Apress. Thank you Steve Anglin, Matthew Moodie, Corbin Collins, Mary Ann Fugate, Adam Heath, Anne Collette, and the rest of the Apress staff for your extraordinary effort.

A huge thank you to Steve Bull and Wallace Jackson, the technical reviewers for testing every piece of code and for filling in the blanks when I missed something. Your contributions were invaluable!

It goes without saying but this book could not have been written if it weren't for the folks responsible for bringing Android into existence. Thank you to them, particularly Dave Sparks from Google who made himself available for some very valuable fact checking and question answering.

To all of my friends and family who were so encouraging, thank you.

Finally, of course, this book would not have happened without the support of my wonderful wife, Karen Van Every. Thank you!

Preface

Among all the things that mobile phones are and have become, one definite trend is the increase in the media production and consumption capabilities they offer. This trend began with the advent of the camera phone in the late 1990s, and over the last few years has dramatically taken off with the surging popularity of smart phones. In terms of media capabilities, today's mobile handsets are simultaneously cameras, photo albums, camcorders, movie players, music players, dictation machines, and potentially much more.

In particular, Android has rich capabilities available within the SDK that this book seeks to illuminate with discussion and examples so that you can get a jump-start on developing the next generation media applications. It walks you through examples that not only show how to display and play media but also allow you to take advantage of the camera, microphone, and video capture capabilities. It is organized more or less into four sections: The first four chapters deal with imaging; the second four handle audio; and the final four are about video and harnessing web services for finding and sharing media.

The examples presented within get a bit more challenging as the book progresses, as the amount of work that needs to be done to develop applications that harness the capabilities increases. Regardless, with some familiarity with Android application development you, the reader should be able to jump to any section and utilize the discussion and example code to create an application that utilizes the capabilities presented.

The examples are generally in the form of a full class that extends an Activity targeted to run with the SDK version 4 (Android 1.6) or later. The examples also include the contents of an XML layout file and in many cases the contents of the `AndroidManifest.xml` file. It is assumed that you will be using Eclipse (Galileo or later) with the ADT plugin (0.9.9 or later) and using the Android SDK (r7 or later). Since much of the book is geared toward audio and video, I advise that you run the examples on a handset (running Android 1.6 or later) rather than on the emulator, because in many cases the examples do not function on the emulator.

I am excited to see what the future of media applications on mobile devices is. It is my hope that through this book I can help you to create and define that future. I look forward to seeing your Android media applications in action.

With all that out of the way, let's get started!

Introduction to Android Imaging

In this chapter, we'll look at the basics of image capture and storage on Android. We'll explore the built-in capabilities that Android provides first and in later chapters move into more custom software. The built-in capabilities for image capture and storage provide a good introduction to the overall media capabilities on Android and pave the way toward what we'll be doing in later chapters with audio and video.

With that in mind, we'll start with how to harness the built-in Camera application and move on to utilizing the MediaStore, the built-in media and metadata storage mechanism. Along the way, we'll look at ways to reduce memory usage and leverage EXIF, the standard in the consumer electronics and image processing software worlds for sharing metadata.

Image Capture Using the Built-In Camera Application

With mobile phones quickly becoming mobile computers, they have in many ways replaced a whole variety of consumer electronics. One of the earliest non-phone related hardware capabilities added to mobile phones was a camera. Currently, it seems someone would be hard pressed to buy a mobile phone that doesn't include a camera. Of course, Android-based phones are no exception; from the beginning, the Android SDK has supported accessing the built-in hardware camera on phones to capture images.

The easiest and most straightforward way to do many things on Android is to leverage an existing piece of software on the device by using an **intent**. An intent is a core component of Android that is described in the documentation as a "description of an action to be performed." In practice, intents are used to trigger other applications to do something or to switch between activities in a single application.

All stock Android devices with the appropriate hardware (camera) come with the Camera application. The Camera application includes an intent filter, which allows developers to

offer image capture capabilities on a par with the Camera application without having to build their own custom capture routines.

An intent filter is a means for a programmer of an application to specify that their application offers a specific capability. Specifying an intent filter in the `AndroidManifest.xml` file of an application tells Android that this application and, in particular, the activity that contains the intent filter will perform the specified task, on command.

The Camera application has the following intent filter specified in its manifest file. The intent filter shown here is contained within the “Camera” activity tags.

```
<intent-filter>
    <action android:name="android.media.action.IMAGE_CAPTURE" />
    <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

In order to utilize the Camera application via an intent, we simply have to construct an intent that will be caught by the foregoing filter.

```
Intent i = new Intent("android.media.action.IMAGE_CAPTURE");
```

In practice, we probably don’t want to create the intent with that action string directly. In this case, a constant is specified in the `MediaStore` class, `ACTION_IMAGE_CAPTURE`. The reason we should use the constant rather than the string itself is that if the string happens to change, it is likely that the constant will change as well, thereby making our call a bit more future-proof than it would otherwise be.

```
Intent i = new Intent(android.provider.MediaStore.ACTION_IMAGE_CAPTURE);
startActivity(i);
```

Using this intent in a basic Android activity will cause the default Camera application to launch in still picture mode, as shown in Figure 1–1.

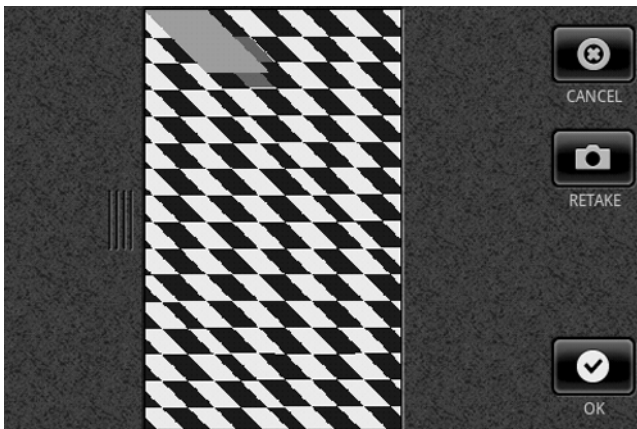


Figure 1–1. The built-in Camera application as called from an intent shown running in an emulator

Returning Data from the Camera App

Of course, simply capturing an image using the built-in camera application won't actually be useful without having the Camera application return the picture to the calling activity when one is captured. This can be accomplished by substituting the `startActivity` method in our activity with the `startActivityForResult` method. Using this method allows us the ability to access the data returned from the Camera application, which happens to be the image that was captured by the user as a `Bitmap`.

Here is a basic example:

```
package com.apress.proandroidmedia.ch1.cameraintent;

import android.app.Activity;
import android.content.Intent;
import android.graphics.Bitmap;
import android.os.Bundle;
import android.widget.ImageView;

public class CameraIntent extends Activity {

    final static int CAMERA_RESULT = 0;

    ImageView imv;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        Intent i = new Intent(android.provider.MediaStore.ACTION_IMAGE_CAPTURE);
        startActivityForResult(i, CAMERA_RESULT);
    }

    protected void onActivityResult(int requestCode, int resultCode, Intent intent) {
        super.onActivityResult(requestCode, resultCode, intent);

        if (resultCode == RESULT_OK)
        {
            Get Bundle extras = intent.getExtras();
            Bitmap bmp = (Bitmap) extras.get("data");

            imv = (ImageView) findViewById(R.id.ReturnedImageView);
            imv.setImageBitmap(bmp);
        }
    }
}
```

It requires the following in the project's `layout/main.xml` file:

```
<?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"
    >
```

```
<ImageView android:id="@+id/ReturnedImageView" android:layout_width="wrap_content"
android:layout_height="wrap_content"></ImageView>
</LinearLayout>
```

To complete the foregoing example, here are the contents of `AndroidManifest.xml`.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="1"
    android:versionName="1.0" package="com.apress.proandroidmedia.ch1.cameraintent">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".CameraIntent"
            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>
    <uses-sdk android:minSdkVersion="4" />
</manifest>
```

In this example, the image is returned from the Camera application in an **extra** passed through the intent that is sent to our calling activity in the `onActivityResult` method. The name of the extra is "data" and it contains a `Bitmap` object, which needs to be cast from a generic `Object`.

```
// Get Extras from the intent
Bundle extras = intent.getExtras();

// Get the returned image from that extra
Bitmap bmp = (Bitmap) extras.get("data");
```

In our layout XML (`layout/main.xml`) file, we have an `ImageView`. An `ImageView` is an extension of a generic `View`, which supports the display of images. Since we have an `ImageView` with the id `ReturnedImageView` specified, in our activity we need to obtain a reference to that and set its `Bitmap` through its `setImageBitmap` method to be our returned image. This enables the user of our application to view the image that was captured.

To get a reference to the `ImageView` object, we use the standard `findViewById` method specified in the `Activity` class. This method allows us to programmatically reference elements specified in the layout XML file that we are using via `setContentView` by passing in the id of the element. In the foregoing example, the `ImageView` object is specified in the XML as follows:

```
<ImageView android:id="@+id/ReturnedImageView" android:layout_width="wrap_content"
android:layout_height="wrap_content"></ImageView>
```

To reference the `ImageView` and tell it to display the `Bitmap` from the Camera, we use the following code.

```
imv = (ImageView) findViewById(R.id.ReturnedImageView);imv.setImageBitmap(bmp);
```

When you run this example, you'll probably notice that the resulting image is small. (On my phone, it is 121 pixels wide by 162 pixels tall. Other devices have different default

sizes.) This is not a bug—rather, it is by design. The Camera application, when triggered via an intent, does not return the full-size image back to the calling activity. In general, doing so would require quite a bit of memory, and the mobile device is generally constrained in this respect. Instead the Camera application returns a small thumbnail image in the returned intent, as shown in Figure 1–2.

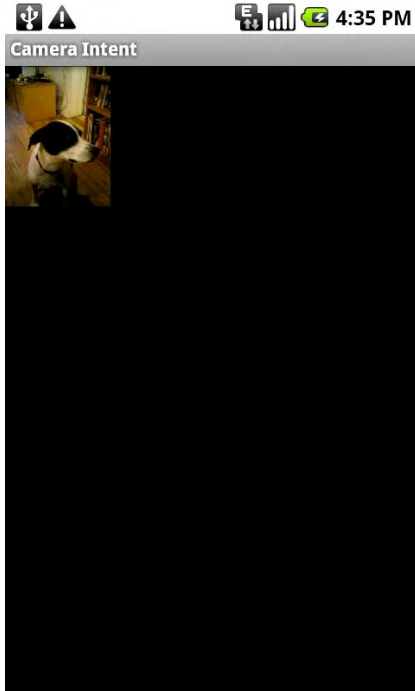


Figure 1–2. The resulting 121x162 pixel image displayed in our ImageView

Capturing Larger Images

To get around the size limitation, starting with Android 1.5, on most devices we can pass an extra into the intent that is used to trigger the Camera application. The name for this extra is specified in the `MediaStore` class as a constant called `EXTRA_OUTPUT`. The value (extras take the form of name-value pairs) for this extra indicates to the Camera application where you would like the captured image saved in the form of an URI.

The following code snippet specifies to the Camera application that the image should be saved to the SD card on a device with a file name of `myfavoritepicture.jpg`.

```
String imageFilePath = Environment.getExternalStorageDirectory().getAbsolutePath()
    + "/myfavoritepicture.jpg";
File imageFile = new File(imageFilePath);
Uri imageFileUri = Uri.fromFile(imageFile);

Intent i = new Intent(android.provider.MediaStore.ACTION_IMAGE_CAPTURE);
i.putExtra(android.provider.MediaStore.EXTRA_OUTPUT, imageFileUri);
startActivityForResult(i, CAMERA_RESULT);
```

NOTE: The foregoing code snippet for creating the URI to the image file could be simplified to the following: `imageFileUri =`

```
Uri.parse("file:///sdcard/myfavoritepicture.jpg");
```

In practice, though, using the method shown will be more device-independent and future-proof should the SD card–naming conventions or the URI syntax for the local filesystem change.

Displaying Large Images

Loading and displaying an image has significant memory usage implications. For instance, the HTC G1 phone has a 3.2-megapixel camera. A 3.2-megapixel camera typically captures images at 2048 pixels by 1536 pixels. Displaying a 32-bit image of that size would take more than 100663kb or approximately 13MB of memory. While this may not guarantee that our application will run out of memory, it will certainly make it more likely.

Android offers us a utility class called `BitmapFactory`, which provides a series of static methods that allow the loading of `Bitmap` images from a variety of sources. For our needs, we'll be loading it from a file to display in our original activity. Fortunately, the methods available in `BitmapFactory` take in a `BitmapFactory.Options` class, which allows us to define how the `Bitmap` is read into memory. Specifically, we can set the sample size that the `BitmapFactory` should use when loading an image. Indicating the `inSampleSize` parameter in `BitmapFactory.Options` indicates that the resulting `Bitmap` image will be that fraction of the size once loaded. For instance, setting the `inSampleSize` to 8 as I do here would yield an image that is 1/8 the size of the original image.

```
BitmapFactory.Options bmpFactoryOptions = new BitmapFactory.Options();  
bmpFactoryOptions.inSampleSize = 8;  
Bitmap bmp = BitmapFactory.decodeFile(imageFilePath, bmpFactoryOptions);  
imv.setImageBitmap(bmp);
```

This is a quick way to load up a large image but doesn't really take into account the image's original size nor the size of the screen. It would be better if we scaled the image to something that would fit nicely on our screen.

The segments of code that follow illustrate how to use the dimensions of the display to determine the amount of down sampling that should occur when loading the image. When we use these methods, the image is assured of filling the bounds of the display as much as possible. If, however, the image is only going to be shown at 100 pixels in any one dimension, that value should be used instead of the display dimensions, which we obtain as follows.

```
Display currentDisplay = getWindowManager().getDefaultDisplay();
int dw = currentDisplay.getWidth();
int dh = currentDisplay.getHeight();
```

To determine the overall dimensions of the image, which are needed for the calculation, we use the `BitmapFactory` and `BitmapFactory.Options` with the `BitmapFactory.Options.inJustDecodeBounds` variable set to `true`. This tells the `BitmapFactory` class to just give us the bounds of the image rather than attempting to decode the image itself. When we use this method, the `BitmapFactory.Options.outHeight` and `BitmapFactory.Options.outWidth` variables are filled in.

```
// Load up the image's dimensions not the image itself
BitmapFactory.Options bmpFactoryOptions = new BitmapFactory.Options();
bmpFactoryOptions.inJustDecodeBounds = true;
Bitmap bmp = BitmapFactory.decodeFile(imageFilePath, bmpFactoryOptions);

int heightRatio = (int)Math.ceil(bmpFactoryOptions.outHeight/(float)dh);
int widthRatio = (int)Math.ceil(bmpFactoryOptions.outWidth/(float)dw);

Log.v("HEIGHTRATIO", ""+heightRatio);
Log.v("WIDTHRATIO", ""+widthRatio);
```

Simple division of the dimensions of the image by the dimensions of the display tells us the ratio. We can then choose whether to use the height ratio or the width ratio, depending on which is greater. Simply using that ratio as the `BitmapFactory.Options.inSampleSize` variable will yield an image that should be loaded into memory with dimensions close to the same dimensions that we need—in this case, close to the dimensions of the display itself.

```
// If both of the ratios are greater than 1,
// one of the sides of the image is greater than the screen
if (heightRatio > 1 && widthRatio > 1)
{
    if (heightRatio > widthRatio)
    {
        // Height ratio is larger, scale according to it
        bmpFactoryOptions.inSampleSize = heightRatio;
    }
    else
    {
        // Width ratio is larger, scale according to it
        bmpFactoryOptions.inSampleSize = widthRatio;
    }
}

// Decode it for real
bmpFactoryOptions.inJustDecodeBounds = false;
bmp = BitmapFactory.decodeFile(imageFilePath, bmpFactoryOptions);
```

Here is the code for a full example that uses the built-in camera via an intent and displays the resulting picture. Figure 1–3 shows a resulting screen sized image as generated by this example.

```
package com.apress.proandroidmedia.ch1.sizedcameraintent;

import java.io.File;

import android.app.Activity;
import android.content.Intent;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.net.Uri;
import android.os.Bundle;
import android.os.Environment;
import android.util.Log;
import android.view.Display;
import android.widget.ImageView;

public class SizedCameraIntent extends Activity {

    final static int CAMERA_RESULT = 0;

    ImageView imv;
    String imageFilePath;

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

        imageFilePath = Environment.getExternalStorageDirectory().getAbsolutePath() +
            "/myfavoritepicture.jpg";
        File imageFile = new File(imageFilePath);
        Uri imageFileUri = Uri.fromFile(imageFile);

        Intent i = new Intent(android.provider.MediaStore.ACTION_IMAGE_CAPTURE);
        i.putExtra(android.provider.MediaStore.EXTRA_OUTPUT, imageFileUri);
        startActivityForResult(i, CAMERA_RESULT);
    }

    protected void onActivityResult(int requestCode, int resultCode, Intent intent) {
        super.onActivityResult(requestCode, resultCode, intent);

        if (resultCode == RESULT_OK)
        {
            // Get a reference to the ImageView
            imv = (ImageView) findViewById(R.id.ReturnedImageView);

            Display currentDisplay = getWindowManager().getDefaultDisplay();
            int dw = currentDisplay.getWidth();
            int dh = currentDisplay.getHeight();

            // Load up the image's dimensions not the image itself
            BitmapFactory.Options bmpFactoryOptions = new BitmapFactory.Options();
            bmpFactoryOptions.inJustDecodeBounds = true;
            Bitmap bmp = BitmapFactory.decodeFile(imageFilePath, bmpFactoryOptions);

            int heightRatio = (int)Math.ceil(bmpFactoryOptions.outHeight/(float)dh);
            int widthRatio = (int)Math.ceil(bmpFactoryOptions.outWidth/(float)dw);
```

```

Log.v("HEIGHTRATIO",""+heightRatio);
Log.v("WIDTHRATIO",""+widthRatio);

// If both of the ratios are greater than 1,
// one of the sides of the image is greater than the screen
if (heightRatio > 1 && widthRatio > 1)
{
    if (heightRatio > widthRatio)
    {
        // Height ratio is larger, scale according to it
        bmpFactoryOptions.inSampleSize = heightRatio;
    }
    else
    {
        // Width ratio is larger, scale according to it
        bmpFactoryOptions.inSampleSize = widthRatio;
    }
}

// Decode it for real
bmpFactoryOptions.inJustDecodeBounds = false;
bmp = BitmapFactory.decodeFile(imageFilePath, bmpFactoryOptions);

// Display it
imv.setImageBitmap(bmp);
}
}
}

```

The foregoing code requires the following layout/main.xml file:

```

<?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"
    >
    <ImageView android:id="@+id/ReturnedImageView" android:layout_width="wrap_content"
    android:layout_height="wrap_content"></ImageView>
</LinearLayout>

```

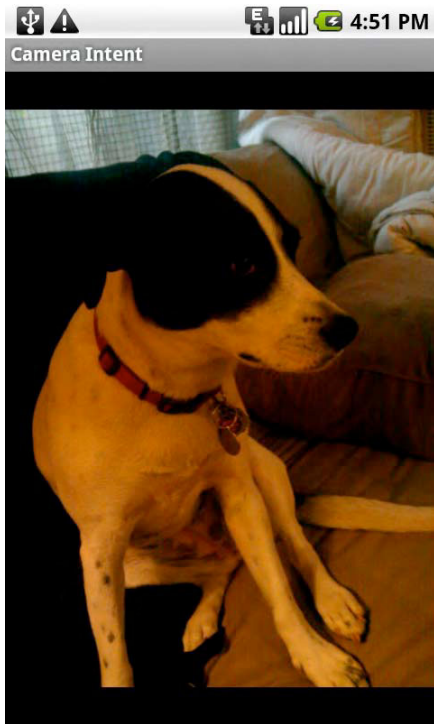


Figure 1–3. *The resulting screen-sized image displayed in our ImageView*

Image Storage and Metadata

Android has a standard way to share data across applications. The classes responsible for this are called **content providers**. Content providers offer a standard interface for the storage and retrieval of various types of data.

The standard content provider for images (as well as audio and video) is the MediaStore. The MediaStore allows the setting of the file in a standard location on the device and has facilities for storing and retrieving metadata about that file. Metadata is data about data; it could include information about the data in the file itself, such as its size and name, but the MediaStore also allows setting for a wide variety of additional data, such as title, description, latitude, and longitude.

To start utilizing the MediaStore, let's change our `SizedCameraIntent` activity so that it uses it for image storage and metadata association instead of storing the image in an arbitrary file on the SD card.

Obtaining an URI for the Image

To obtain the standard location for storage of images, we first need to get a reference to the `MediaStore`. To do this, we use a **content resolver**. A content resolver is the means to access a content provider, which the `MediaStore` is.

By passing a specific URI, the content resolver knows to provide an interface to the `MediaStore` as the content provider. Since we are inserting a new image, the method we are using is `insert` and the URI that we should use is contained in a constant in the `android.provider.MediaStore.Images.Media` class called `EXTERNAL_CONTENT_URI`. This means that we want to store the image on the primary external volume of the device, generally the SD card. If we wanted to store it instead in the internal memory of the device, we could use `INTERNAL_CONTENT_URI`. Generally, though, for media storage, as images, audio, and video can be rather large in size, you'll want to use the `EXTERNAL_CONTENT_URI`.

The `insert` call shown previously returns an URI, which we can use to write the image file's binary data to. In our case, as we are doing in the `CameraActivity`, we want to simply pass that as an extra in the intent that triggers the Camera application.

```
Uri imageFileUri = getContentResolver().insert(
    Media.EXTERNAL_CONTENT_URI, new ContentValues());

Intent i = new Intent(android.provider.MediaStore.ACTION_IMAGE_CAPTURE);
i.putExtra(android.provider.MediaStore.EXTRA_OUTPUT, imageFileUri);
startActivityForResult(i, CAMERA_RESULT);
```

You'll notice that we are also passing in a new `ContentValues` object. The `ContentValues` object is the metadata that we want to associate with the record when it is created. In the preceding example, we are passing in an empty `ContentValues` object.

Prepopulating Associated Metadata

If we wanted to pre-fill the metadata, we would use the `put` method to add some data into it. `ContentValues` takes data as name-value pairs. The names are standard and defined as constants in the `android.provider.MediaStore.Images.Media` class. (Some of the constants are actually located in the `android.provider.MediaStore.MediaColumns` interface, which the `Media` class implements.)

```
// Save the name and description of an image in a ContentValues map.
ContentValues contentValues = new ContentValues(3);
contentValues.put(Media.DISPLAY_NAME, "This is a test title");
contentValues.put(Media.DESCRPTION, "This is a test description");
contentValues.put(Media.MIME_TYPE, "image/jpeg");

// Add a new record without the bitmap, but with some values set.
// insert() returns the URI of the new record.
Uri imageFileUri = getContentResolver().insert(Media.EXTERNAL_CONTENT_URI, ←
    contentValues);
```

Again, what is returned by this call is a URI that can be passed to the Camera application via the intent to specify the location that the image should be saved in.

If you output this URI via a Log command, it should look something like this:

```
content://media/external/images/media/16
```

The first thing you might notice is that it looks like a regular URL, such as you would use in a web browser; but instead of starting with something like `http`, which is the protocol that delivers web pages, it starts with `content`. In Android, when a URI starts with `content`, it is one that is used with a content provider (such as `MediaStore`).

Retrieving the Saved Image

The same URI obtained previously for saving the image can be used as the means to access the image as well. Instead of passing in the full path to the file to our `BitmapFactory`, we can instead open an `InputStream` for the image via the content resolver and pass that to `BitmapFactory`.

```
Bitmap bmp = BitmapFactory.decodeStream(↵  
getContentResolver().openInputStream(imageFileUri), null, bmpFactoryOptions);
```

Adding Metadata Later

If we want to associate more metadata with the image after we have captured it into the `MediaStore`, we can use the `update` method of our content resolver. This is very similar to the `insert` method we used previously, except we are accessing the image file directly with the URI to the image file.

```
// Update the record with Title and Description  
ContentValues contentValues = new ContentValues(3);  
contentValues.put(Media.DISPLAY_NAME, "This is a test title");  
contentValues.put(Media.DESCRPTION, "This is a test description");  
getContentResolver().update(imageFileUri, contentValues, null, null);
```

Updating Our CameraActivity to Use MediaStore for Image Storage and to Associate Metadata

The following is an update to our previous example, which saves our image in the `MediaStore` and then presents us with an opportunity to add a title and description. In addition, this version has several UI elements whose visibility is managed based upon the progress of the user in the application.

```
package com.apress.proandroidmedia.ch1.mediastorecameraintent;  
  
import java.io.FileNotFoundException;  
import android.app.Activity;  
import android.content.Intent;  
import android.graphics.Bitmap;  
import android.graphics.BitmapFactory;  
import android.net.Uri;
```



```

import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ImageView;
import android.widget.TextView;
import android.widget.Toast;
import android.provider.MediaStore.Images.Media;
import android.content.ContentValues;

public class MediaStoreCameraIntent extends Activity {

    final static int CAMERA_RESULT = 0;

    Uri imageFileUri;

    // User interface elements, specified in res/layout/main.xml
    ImageView returnedImageView;
    Button takePictureButton;
    Button saveDataButton;
    TextView titleTextView;
    TextView descriptionTextView;
    EditText titleEditText;
    EditText descriptionEditText;

```

We are including a couple of user interface elements. They are specified as normal in `layout/main.xml` and their objects are declared in the foregoing code.

```

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

    // Set the content view to be what is defined in the res/layout/main.xml file
    setContentView(R.layout.main);

    // Get references to UI elements
    returnedImageView = (ImageView) findViewById(R.id.ReturnedImageView);
    takePictureButton = (Button) findViewById(R.id.TakePictureButton);
    saveDataButton = (Button) findViewById(R.id.SaveDataButton);
    titleTextView = (TextView) findViewById(R.id.TitleTextView);
    descriptionTextView = (TextView) findViewById(R.id.DescriptionTextView);
    titleEditText = (EditText) findViewById(R.id.TitleEditText);
    descriptionEditText = (EditText) findViewById(R.id.DescriptionEditText);

```

In the standard activity `onCreate` method, after we call `setContentView`, we instantiate the user interface elements that we'll need control over in code. We have to cast each one to the appropriate type after obtaining it via the `findViewById` method.

```

// Set all except takePictureButton to not be visible initially
// View.GONE is invisible and doesn't take up space in the layout
returnedImageView.setVisibility(View.GONE);
saveDataButton.setVisibility(View.GONE);
titleTextView.setVisibility(View.GONE);
descriptionTextView.setVisibility(View.GONE);
titleEditText.setVisibility(View.GONE);
descriptionEditText.setVisibility(View.GONE);

```

Continuing on, we set all of the user interface elements to not be visible and not to take up space in the layout. `View.GONE` is the constant that can be used in the `setVisibility` method to do this. The other option, `View.INVISIBLE`, hides them but they still take up space in the layout.

```
// When the Take Picture Button is clicked
takePictureButton.setOnClickListener(new OnClickListener() {
    public void onClick(View v)
    {
        // Add a new record without the bitmap
        // returns the URI of the new record
        imageFileUri = getContentResolver().insert(Media.EXTERNAL_CONTENT_URI, ←
new ContentValues());

        // Start the Camera App
        Intent i = new Intent(android.provider.MediaStore.ACTION_IMAGE_CAPTURE);
        i.putExtra(android.provider.MediaStore.EXTRA_OUTPUT, imageFileUri);
        startActivityForResult(i, CAMERA_RESULT);
    }
});
```

In the `OnClickListener` for the `takePictureButton`, we create the standard intent for the built-in camera and call `startActivityForResult`. Doing it here rather than directly in the `onCreate` method makes for a slightly nicer user experience.

```
saveDataButton.setOnClickListener(new OnClickListener() {
    public void onClick(View v)
    {
        // Update the MediaStore record with Title and Description
        ContentValues contentValues = new ContentValues(3);
        contentValues.put(Media.DISPLAY_NAME, ←
titleEditText.getText().toString());
        contentValues.put(Media.DESCRPTION, ←
descriptionEditText.getText().toString());
        getContentResolver().update(imageFileUri, contentValues, null, null);

        // Tell the user
        Toast bread = Toast.makeText(MediaStoreCameraIntent.this, "Record ←
Updated", Toast.LENGTH_SHORT);
        bread.show();

        // Go back to the initial state, set Take Picture Button Visible
        // hide other UI elements
        takePictureButton.setVisibility(View.VISIBLE);

        returnedImageView.setVisibility(View.GONE);
        saveDataButton.setVisibility(View.GONE);
        titleTextView.setVisibility(View.GONE);
        descriptionTextView.setVisibility(View.GONE);
        titleEditText.setVisibility(View.GONE);
        descriptionEditText.setVisibility(View.GONE);
    }
});
}
```

The `OnClickListener` for the `saveDataButton`, which is visible once the Camera application has returned an image, does the work of associating the metadata with the image. It takes the values that the user has typed into the various `EditText` elements and creates a `ContentValues` object that is used to update the record for this image in the `MediaStore`.

```
protected void onActivityResult(int requestCode, int resultCode, Intent intent)
{
    super.onActivityResult(requestCode, resultCode, intent);

    if (resultCode == RESULT_OK)
    {
        // The Camera App has returned

        // Hide the Take Picture Button
        takePictureButton.setVisibility(View.GONE);

        // Show the other UI Elements
        saveDataButton.setVisibility(View.VISIBLE);
        returnedImageView.setVisibility(View.VISIBLE);
        titleTextView.setVisibility(View.VISIBLE);
        descriptionTextView.setVisibility(View.VISIBLE);
        titleEditText.setVisibility(View.VISIBLE);
        descriptionEditText.setVisibility(View.VISIBLE);

        // Scale the image
        int dw = 200; // Make it at most 200 pixels wide
        int dh = 200; // Make it at most 200 pixels tall

        try
        {
            // Load up the image's dimensions not the image itself
            BitmapFactory.Options bmpFactoryOptions = new BitmapFactory.Options();
            bmpFactoryOptions.inJustDecodeBounds = true;
            Bitmap bmp = BitmapFactory.decodeStream(getContentResolver().
openInputStream(imageFileUri), null, bmpFactoryOptions);

            int heightRatio = (int)Math.ceil(bmpFactoryOptions.outHeight/(float)dh);
            int widthRatio = (int)Math.ceil(bmpFactoryOptions.outWidth/(float)dw);

            Log.v("HEIGHTRATIO", ""+heightRatio);
            Log.v("WIDTHRATIO", ""+widthRatio);

            // If both of the ratios are greater than 1,
            // one of the sides of the image is greater than the screen
            if (heightRatio > 1 && widthRatio > 1)
            {
                if (heightRatio > widthRatio)
                {
                    // Height ratio is larger, scale according to it
                    bmpFactoryOptions.inSampleSize = heightRatio;
                }
                else
                {
                    // Width ratio is larger, scale according to it
                    bmpFactoryOptions.inSampleSize = widthRatio;
                }
            }
        }
    }
}
```


applications will use the default MediaStore, we can leverage that to build our own gallery application.

Selecting from the MediaStore is very straightforward. We use the same URI that we used to create a new record, to select records from it.

```
Media.EXTERNAL_CONTENT_URI
```

The MediaStore and, in fact, all content providers operate in a similar manner to a database. We select records from them and are given a Cursor object, which we can use to iterate over the results.

In order to do the selection in the first place, we need to create a string array of the columns we would like returned. The standard columns for images in the MediaStore are represented in the MediaStore.Images.Media class.

```
String[] columns = { Media.DATA, Media._ID, Media.TITLE, Media.DISPLAY_NAME };
```

To perform the actual query, we can use the activity managedQuery method. The first argument is the URI, followed by the array of column names, followed by a limiting WHERE clause, any arguments for the WHERE clause, and, lastly, an ORDER BY clause.

The following would select records that were created within the last hour and order them oldest to most recent.

First we create a variable called oneHourAgo, which holds the number of seconds elapsed from January 1, 1970 as of one hour ago. System.currentTimeMillis() returns the number of milliseconds from the same date, so dividing by 1000 gives us the number of seconds. If we subtract 60 minutes * 60 seconds, we'll get the value as of one hour ago.

```
long oneHourAgo = System.currentTimeMillis()/1000 - (60 * 60);
```

We then place that value in an array of strings that we can use as the arguments for the WHERE clause.

```
String[] whereValues = {""+oneHourAgo};
```

Then we choose the columns we want returned.

```
String[] columns = { Media.DATA, Media._ID, Media.TITLE, Media.DISPLAY_NAME, ↵  
Media.DATE_ADDED };
```

And finally we perform the query. The WHERE clause has a ?, which will get substituted with the value in the next parameter. If there are multiple ?, there must be multiple values in the array passed in. The ORDER BY clause used here specifies that the data returned will be ordered by the date added in ascending order.

```
cursor = managedQuery(Media.EXTERNAL_CONTENT_URI, columns, Media.DATE_ADDED + " > ?", ↵  
whereValues, Media.DATE_ADDED + " ASC");
```

You can, of course, pass in null for the last three arguments if you want all records returned.

```
Cursor cursor = managedQuery(Media.EXTERNAL_CONTENT_URI, columns, null, null, null);
```

The cursor returned can tell us the index of each of the columns as selected.

```
displayColumnIndex = cursor.getColumnIndexOrThrow(MediaStore.Images.Media.DATA);
```

We need the index in order to select that field out of the cursor. First we make sure that the cursor is valid and has some results by calling the `moveToFirst` method. This method will be false if the cursor isn't holding any results. We use one of several methods in the `Cursor` class to select the actual data. The method we choose is dependent on what type the data is, `getString` for strings, `getInt` for integers, and so on.

```
if (cursor.moveToFirst()) {
    String displayName = cursor.getString(displayColumnIndex);
}
```

Creating an Image Viewing Application

What follows is a full example that queries the `MediaStore` to find images and presents them to the user one after the other in the form of a slideshow.

```
package com.apress.proandroidmedia.ch1.mediastoregallery;

import android.app.Activity;
import android.database.Cursor;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.os.Bundle;
import android.provider.MediaStore;
import android.provider.MediaStore.Images.Media;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.ImageButton;
import android.widget.TextView;
```

```
public class MediaStoreGallery extends Activity {

    public final static int DISPLAYWIDTH = 200;
    public final static int DISPLAYHEIGHT = 200;
```

Instead of using the size of the screen to load and display the images, we'll use the foregoing constants to decide how large to display them.

```
    TextView titleTextView;
    ImageButton imageButton;
```

In this example, we are using an `ImageButton` instead of an `ImageView`. This gives us both the functionality of a `Button` (which can be clicked) and an `ImageView` (which can display an image).

```
    Cursor cursor;
    Bitmap bmp;
    String imageFilePath;
    int fileColumn;
    int titleColumn;
    int displayColumn;

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