16
Taking Pictures with Intents

Now that you know how to work with implicit intents, you can document your crimes in even more detail. With a picture of the crime, you can share the gory details with everyone.

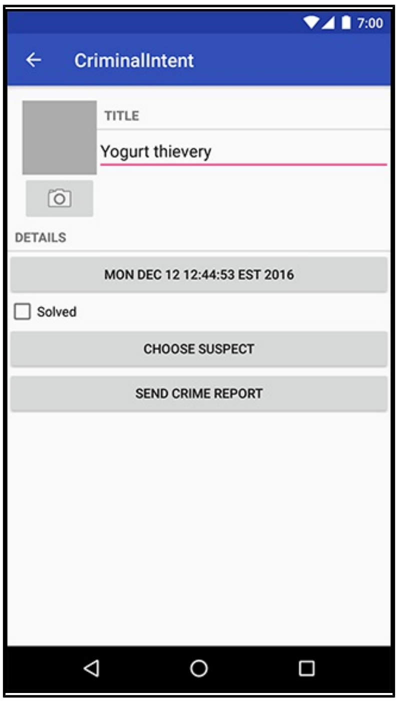
Taking a picture will involve a couple of new tools, used in combination with a tool you recently got to know: the implicit intent. An implicit intent can be used to start up the user's favorite camera application and receive a new picture from it.

An implicit intent can get you a picture, but where do you put it? And once the picture comes in, how do you display it? In this chapter, you will answer both of those questions.

A Place for Your Photo

The first step is to build out a place for your photo to live. You will need two new **`View`** objects: an **`ImageView`** to display the photo and a **`Button`** to press to take a new photo (Figure 16.1).

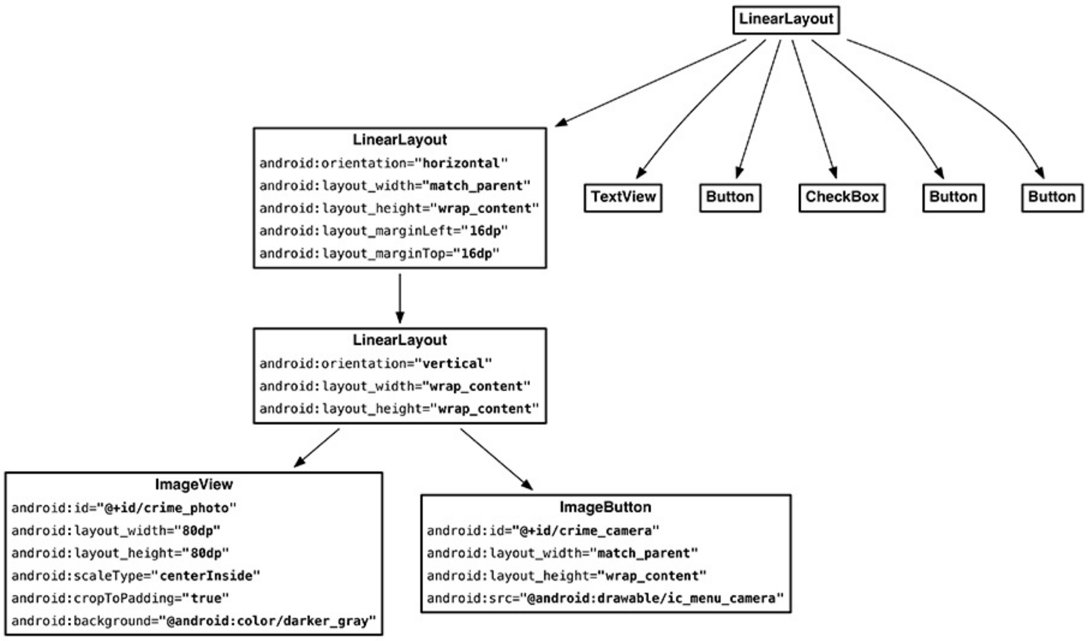
Figure 16.1 New UI



Dedicating an entire row to a thumbnail and a button would make your app look clunky and unprofessional. You do not want that, so you will arrange things nicely.

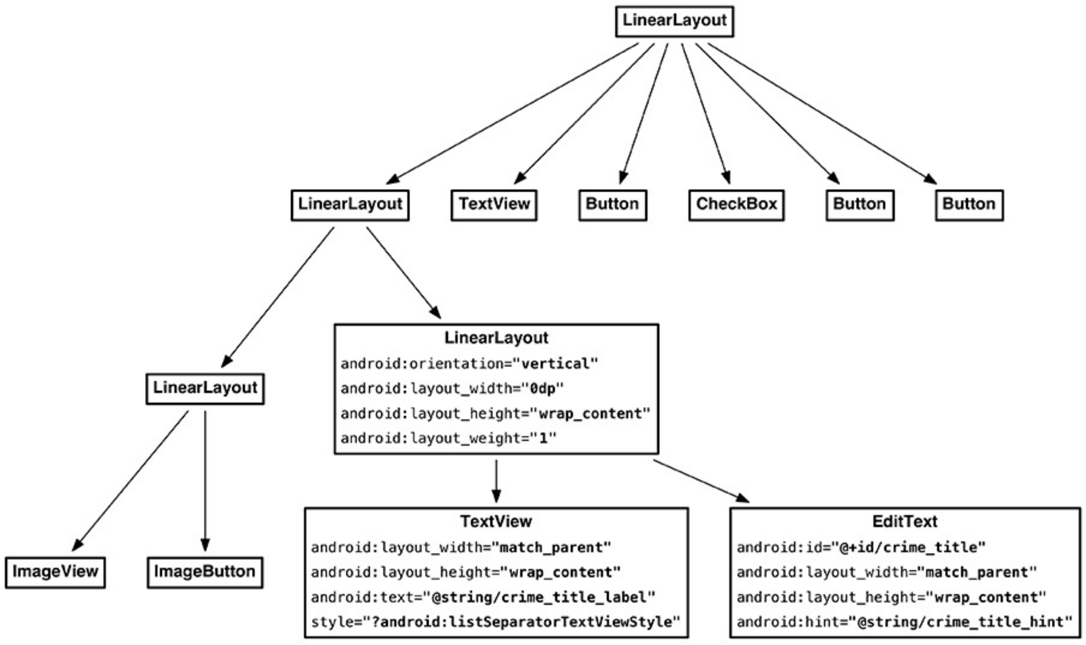
Add new views to `fragment_crime.xml` to build out this new area. Start with the lefthand side, adding an **`ImageView`** for the picture and an **`ImageButton`** to take a picture (Figure 16.2).

Figure 16.2 Camera view layout (`res/layout/fragment_crime.xml`)


LinearLayout
LinearLayout
android:orientation="horizontal"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:layout_marginLeft="16dp"
android:layout_marginTop="16dp"
TextView
Button
CheckBox
Button
Button
LinearLayout
android:orientation="vertical"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
ImageView
android:id="@+id/crime_photo"
android:layout_width="80dp"
android:layout_height="80dp"
android:scaleType="centerInside"
android:cropToPadding="true"
android:background="@android:color/darker_gray"
ImageButton
android:id="@+id/crime_camera"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:src="@android:drawable/ic_menu_camera"

Then continue with the righthand side, moving your title **`TextView`** and **`EditText`** into a new **`LinearLayout`** child to the **`LinearLayout`** you built in Figure 16.2 (Figure 16.3).

Figure 16.3 Title layout (`res/layout/fragment_crime.xml`)


LinearLayout
LinearLayout
TextView
Button
CheckBox
Button
Button
LinearLayout
ImageView
ImageButton
LinearLayout
android:orientation="vertical"
android:layout_width="0dp"
android:layout_height="wrap_content"
android:layout_weight="1"
TextView
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:text="@string/crime_title_label"
style="?android:listSeparatorTextViewStyle"
EditText
android:id="@+id/crime_title"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:hint="@string/crime_title_hint"

Run CriminalIntent, and you should see your new UI looking just like Figure 16.1.

Looks great. Now, to respond to presses on your **`ImageButton`** and to control the content of your **`ImageView`**, you need instance variables referring to each of them. Call **`findViewById(int)`** as usual on your inflated `fragment_crime.xml` to find your new views and wire them up.

Listing 16.1 Adding instance variables (`CrimeFragment.java`)

```
private Button mSuspectButton;
private Button mReportButton;
private ImageButton mPhotoButton;
private ImageView mPhotoView;
...
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
    ...
    PackageManager packageManager = getActivity().getPackageManager();
    if (packageManager.resolveActivity(pickContact,
            PackageManager.MATCH_DEFAULT_ONLY) == null) {
        mSuspectButton.setEnabled(false);
    }

    mPhotoButton = (ImageButton) v.findViewById(R.id.crime_camera);
    mPhotoView = (ImageView) v.findViewById(R.id.crime_photo);

    return v;
}
```

And with that, you are done with the UI for the time being. (You will wire those buttons up in a minute or two.)

File Storage

Your photo needs more than a place on the screen. Full-size pictures are too large to stick inside a SQLite database, much less an **`Intent`**. They will need a place to live on your device's filesystem.

Luckily, you have a place to stash these files: your private storage. Recall that you used your private storage to save your SQLite database. With methods like **`Context.getFileStreamPath(String)`** and **`Context.getFilesDir()`**, you can do the same thing with regular files, too (which will live in a subfolder adjacent to the `databases` subfolder your SQLite database lives in).

These are the basic file and directory methods in the **`Context`** class:

`File` **`getFilesDir()`**

returns a handle to the directory for private application files

`FileInputStream` **`openFileInput(String name)`**

opens an existing file for input (relative to the files directory)

`FileOutputStream` **`openFileOutput(String name, int mode)`**

opens a file for output, possibly creating it (relative to the files directory)

`File` **`getDir(String name, int mode)`**

gets (and possibly creates) a subdirectory within the files directory

`String[]` **`fileList()`**

gets a list of file names in the main files directory, such as for use with **`openFileInput(String)`**

`File` **`getCacheDir()`**

returns a handle to a directory you can use specifically for storing cache files; you should take care to keep this directory tidy and use as little space as possible

There is a catch, though. Because these files are private, *only your own application* can read or write to them. As long as no other app needs to access those files, these methods are sufficient.

However, they are not sufficient if another application needs to write to your files. This is the case for CriminalIntent: The external camera app will need to save the picture it takes as a file in your app. In those cases, these methods do not go far enough: While there is a `Context.MODE_WORLD_READABLE` flag you can pass into **`openFileOutput(String, int)`**, it is deprecated and not completely reliable in its effects on newer devices. Once upon a time you could also transfer files using publicly accessible external storage, but this has been locked down in recent versions of Android for security reasons.

If you need to share or receive files with other apps (files like stored pictures), you need to expose those files through a **`ContentProvider`**. A **`ContentProvider`** allows you to expose content URIs to other apps. They can then download from or write to those content URIs. Either way, you are in control and always have the option to deny those reads or writes if you so choose.

Using FileProvider

When all you need to do is receive a file from another application, implementing an entire **ContentProvider** is overkill. Fortunately, Google has provided a convenience class called **FileProvider** that takes care of everything except the configuration work.

The first step is to declare **FileProvider** as a **ContentProvider** hooked up to a specific *authority*. This is done by adding a content provider declaration to your AndroidManifest.xml.

Listing 16.2 Adding a FileProvider declaration (AndroidManifest.xml)

```
<activity
    android:name=".CrimePagerActivity"
    android:parentActivityName=".CrimeListActivity">
</activity>
<provider
    android:name="android.support.v4.content.FileProvider"
    android:authorities="com.bignerdranch.android.criminalintent.fileprovider"
    android:exported="false"
    android:grantUriPermissions="true">
</provider>
```

The authority is a location – a place that files will be saved to. By hooking up **FileProvider** to your authority, you give other apps a target for their requests. By adding the exported="false" attribute, you keep anyone from using your provider except you or anyone you grant permission to. And by adding the grantUriPermissions attribute, you add the ability to grant other apps permission to write to URIs on this authority when you send them out in an intent. (Keep an eye out for this later.)

Now that you have told Android where your **FileProvider** is, you also need to tell your **FileProvider** which files it is exposing. This bit of configuration is done with an extra XML resource file. Right-click your app/res folder in the project tool window and select New → Android resource file. For Resource type, select XML, and then enter files for the name.

Crack open xml/files.xml, switch to the Text tab, and replace its contents with the following:

Listing 16.3 Filling out the paths description (res/xml/files.xml)

```
~~<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">~~

~~</PreferenceScreen>~~
<paths>
    <files-path name="crime_photos" path="."/>
</paths>
```

This XML file is a description that says, "Map the root path of my private storage as crime_photos." You will not use the crime_photos name – **FileProvider** uses that internally.

Now hook up files.xml to your **FileProvider** by adding a meta-data tag in your AndroidManifest.xml.

Listing 16.4 Hooking up the paths description (AndroidManifest.xml)

```
<provider
    android:name="android.support.v4.content.FileProvider"
    android:authorities="com.bignerdranch.android.criminalintent.fileprovider"
    android:exported="false"
    android:grantUriPermissions="true">
    <meta-data
        android:name="android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/files"/>
</provider>
```

Designating a picture location

Time to give your pictures a place to live locally. First, add a method to **Crime** to get a well-known filename.

Listing 16.5 Adding the filename-derived property (Crime.java)

```
    public void setSuspect(String suspect) {
        mSuspect = suspect;
    }

    public String getPhotoFilename() {
        return "IMG_" + getId().toString() + ".jpg";
    }
}
```

Crime.getPhotoFilename() will not know what folder the photo will be stored in. However, the filename will be unique, since it is based on the **Crime**'s ID.

Next, find where the photos should live. **CrimeLab** is responsible for everything related to persisting data in CriminalIntent, so it is a natural owner for this idea. Add a **getPhotoFile(Crime)** method to **CrimeLab** that provides a complete local filepath for **Crime**'s image.

Listing 16.6 Finding photo file location (CrimeLab.java)

```
public class CrimeLab {
    ...
    public Crime getCrime(UUID id) {
        ...
    }

    public File getPhotoFile(Crime crime) {
        File filesDir = mContext.getFilesDir();
        return new File(filesDir, crime.getPhotoFilename());
    }

    public void updateCrime(Crime crime) {
        ...
    }
```

This code does not create any files on the filesystem. It only returns **File** objects that point to the right locations. Later on, you will use **FileProvider** to expose these paths as URIs.

Using a Camera Intent

The next step is to actually take the picture. This is the easy part: You get to use an implicit intent again.

Start by stashing the location of the photo file. (You will use it a few more times, so this will save a bit of work.)

Listing 16.7 Grabbing photo file location (`CrimeFragment.java`)

```
private Crime mCrime;
private File mPhotoFile;
private EditText mTitleField;
...
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    UUID crimeId = (UUID) getArguments().getSerializable(ARG_CRIME_ID);
    mCrime = CrimeLab.get(getActivity()).getCrime(crimeId);
    mPhotoFile = CrimeLab.get(getActivity()).getPhotoFile(mCrime);
}
```

Next you will hook up the camera button to actually take the picture. The camera intent is defined in **MediaStore**, Android's lord and master of all things media related. You will send an intent with an action of `MediaStore.ACTION_IMAGE_CAPTURE`, and Android will fire up a camera activity and take a picture for you.

But hold that thought for one minute.

Firing the intent

Now you are ready to fire the camera intent. The action you want is called `ACTION_IMAGE_CAPTURE`, and it is defined in the **MediaStore** class. **MediaStore** defines the public interfaces used in Android for interacting with common media – images, videos, and music. This includes the image capture intent, which fires up the camera.

By default, `ACTION_IMAGE_CAPTURE` will dutifully fire up the camera application and take a picture, but it will not be a full-resolution picture. Instead, it will take a small-resolution thumbnail picture and stick it inside the **Intent** object returned in **onActivityResult(…)**.

For a full-resolution output, you need to tell it where to save the image on the filesystem. This can be done by passing a **Uri** pointing to where you want to save the file in `MediaStore.EXTRA_OUTPUT`. This **Uri** will point at a location serviced by **FileProvider**.

Write an implicit intent to ask for a new picture to be taken into the location saved in `mPhotoFile`. Add code to ensure that the button is disabled if there is no camera app or if there is no location at which to save the photo. (To determine whether there is a camera app available, you will query **PackageManager** for activities that respond to your camera implicit intent. Querying the **PackageManager** is discussed in more detail in the section called *Checking for responding activities* in Chapter 15.)

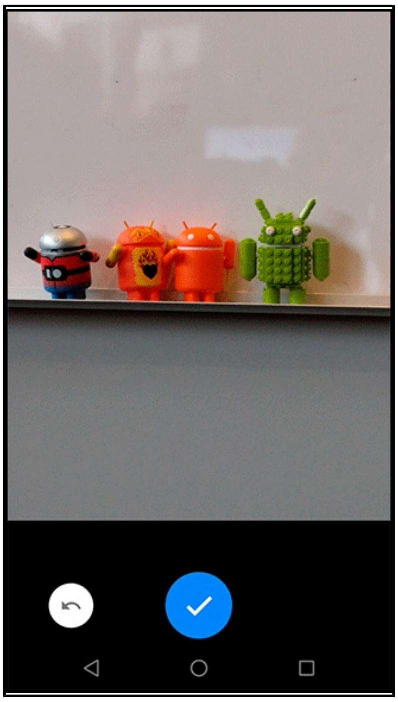
Listing 16.8 Firing a camera intent (CrimeFragment.java)

```
private static final int REQUEST_DATE = 0;
private static final int REQUEST_CONTACT = 1;
private static final int REQUEST_PHOTO= 2;
...
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
    ...
    mPhotoButton = (ImageButton) v.findViewById(R.id.crime_camera);
    final Intent captureImage = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);

    boolean canTakePhoto = mPhotoFile != null &&
            captureImage.resolveActivity(packageManager) != null;
    mPhotoButton.setEnabled(canTakePhoto);

    mPhotoButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            Uri uri = FileProvider.getUriForFile(getActivity(),
                    "com.bignerdranch.android.criminalintent.fileprovider",
                    mPhotoFile);
            captureImage.putExtra(MediaStore.EXTRA_OUTPUT, uri);

            List<ResolveInfo> cameraActivities = getActivity()
                    .getPackageManager().queryIntentActivities(captureImage,
                            PackageManager.MATCH_DEFAULT_ONLY);

            for (ResolveInfo activity : cameraActivities) {
                getActivity().grantUriPermission(activity.activityInfo.packageName,
                        uri, Intent.FLAG_GRANT_WRITE_URI_PERMISSION);
            }

            startActivityForResult(captureImage, REQUEST_PHOTO);
        }
    });

    mPhotoView = (ImageView) v.findViewById(R.id.crime_photo);

    return v;
}
```

Calling **FileProvider.getUriForFile(…)** translates your local filepath into a **Uri** the camera app can see. To actually write to it, though, you need to grant the camera app permission. To do this, you grant the Intent.FLAG_GRANT_WRITE_URI_PERMISSION flag to every activity your cameraImage intent can resolve to. That grants them all a write permission specifically for this one **Uri**. Adding the android:grantUriPermissions attribute in your provider declaration was necessary to open this bit of functionality. Later, you will revoke this permission to close up that gap in your armor again.

Run CriminalIntent and press the camera button to run your camera app (Figure 16.4).

Figure 16.4 [Insert your camera app here]



Scaling and Displaying Bitmaps

With that, you are successfully taking pictures. Your image will be saved to a file on the filesystem for you to use.

Your next step will be to take this file, load it up, and show it to the user. To do this, you need to load it into a reasonably sized **Bitmap** object. To get a **Bitmap** from a file, all you need to do is use the **BitmapFactory** class:

```
Bitmap bitmap = BitmapFactory.decodeFile(mPhotoFile.getPath());
```

There has to be a catch, though, right? Otherwise we would have put that in bold, you would have typed it in, and you would be done.

Here is the catch: When we say "reasonably sized," we mean it. A **Bitmap** is a simple object that stores literal pixel data. That means that even if the original file was compressed, there is no compression in the **Bitmap** itself. So a 16-megapixel, 24-bit camera image – which might only be a 5 MB JPG – would blow up to 48 MB loaded into a **Bitmap** object (!).

You can get around this, but it does mean that you will need to scale the bitmap down by hand. You will first scan the file to see how big it is, next figure out how much you need to scale it by to fit it in a given area, and finally reread the file to create a scaled-down **Bitmap** object.

Create a new class called `PictureUtils.java` for your new method and add a static method to it called **`getScaledBitmap(String, int, int)`**.

Listing 16.9 Creating **getScaledBitmap(…)** (PictureUtils.java)

```
public class PictureUtils {
    public static Bitmap getScaledBitmap(String path, int destWidth, int destHeight) {
        // Read in the dimensions of the image on disk
        BitmapFactory.Options options = new BitmapFactory.Options();
        options.inJustDecodeBounds = true;
        BitmapFactory.decodeFile(path, options);

        float srcWidth = options.outWidth;
        float srcHeight = options.outHeight;

        // Figure out how much to scale down by
        int inSampleSize = 1;
        if (srcHeight > destHeight || srcWidth > destWidth) {
            float heightScale = srcHeight / destHeight;
            float widthScale = srcWidth / destWidth;

            inSampleSize = Math.round(heightScale > widthScale ? heightScale :
                    widthScale);
        }

        options = new BitmapFactory.Options();
        options.inSampleSize = inSampleSize;

        // Read in and create final bitmap
        return BitmapFactory.decodeFile(path, options);
    }
}
```

The key parameter above is `inSampleSize`. This determines how big each "sample" should be for each pixel – a sample size of `1` has one final horizontal pixel for each horizontal pixel in the original file, and a sample size of `2` has one horizontal pixel for every two horizontal pixels in the original file. So when `inSampleSize` is `2`, the image has a quarter of the number of pixels of the original.

One more bit of bad news: When your fragment initially starts up, you will not know how big **PhotoView** is. Until a layout pass happens, views do not have dimensions onscreen. The first layout pass happens after **onCreate(…)**, **onStart()**, and **onResume()** initially run, which is why **PhotoView** does not know how big it is.

There are two solutions to this problem: Either you wait until a layout pass happens, or you use a conservative estimate. The conservative estimate approach is less efficient, but more straightforward. Write another static method called **getScaledBitmap(String, Activity)** to scale a **Bitmap** for a particular **Activity**'s size.

Listing 16.10 Writing conservative scale method (PictureUtils.java)

```
public class PictureUtils {
    public static Bitmap getScaledBitmap(String path, Activity activity) {
        Point size = new Point();
        activity.getWindowManager().getDefaultDisplay()
                .getSize(size);

        return getScaledBitmap(path, size.x, size.y);
    }
```

This method checks to see how big the screen is and then scales the image down to that size. The **ImageView** you load into will always be smaller than this size, so this is a very conservative estimate.

Next, to load this **Bitmap** into your **ImageView**, add a method to **CrimeFragment** to update `mPhotoView`.

Listing 16.11 Updating `mPhotoView` (`CrimeFragment.java`)

```
    private String getCrimeReport() {
        ...
    }

    private void updatePhotoView() {
        if (mPhotoFile == null || !mPhotoFile.exists()) {
            mPhotoView.setImageDrawable(null);
        } else {
            Bitmap bitmap = PictureUtils.getScaledBitmap(
                    mPhotoFile.getPath(), getActivity());
            mPhotoView.setImageBitmap(bitmap);
        }
    }
}
```

Then call that method from inside **onCreateView(…)** and **onActivityResult(…)**.

Listing 16.12 Calling **updatePhotoView()** (`CrimeFragment.java`)

```
    mPhotoButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            ...
            startActivityForResult(captureImage, REQUEST_PHOTO);
        }
    });

    mPhotoView = (ImageView) v.findViewById(R.id.crime_photo);
    updatePhotoView();

    return v;
}

@Override
public void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (resultCode != Activity.RESULT_OK) {
        return;
    }

    if (requestCode == REQUEST_DATE) {
        ...
    } else if (requestCode == REQUEST_CONTACT && data != null) {
        ...
    } else if (requestCode == REQUEST_PHOTO) {
        Uri uri = FileProvider.getUriForFile(getActivity(),
                "com.bignerdranch.android.criminalintent.fileprovider",
                mPhotoFile);

        getActivity().revokeUriPermission(uri,
                Intent.FLAG_GRANT_WRITE_URI_PERMISSION);

        updatePhotoView();
    }
}
```

Now that the camera is done writing to your file, you can revoke the permission, closing off access to your file again. Run CriminalIntent again, and you should see your image displayed in the thumbnail view.

Declaring Features

Your camera implementation works great now. One more task remains: Tell potential users about it. When your app uses a feature like the camera – or near-field communication, or any other feature that may vary from device to device – it is strongly recommended that you tell Android about it. This allows other apps (like the Google Play Store) to refuse to install your app if it uses a feature the device does not support.

To declare that you use the camera, add a `<uses-feature>` tag to your `AndroidManifest.xml`.

Listing 16.13 Adding uses-feature tag (`AndroidManifest.xml`)

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.bignerdranch.android.criminalintent" >

    <uses-feature android:name="android.hardware.camera"
                  android:required="false"
        />
```

You include the optional attribute `android:required` here. Why? By default, declaring that you use a feature means that your app will not work correctly without that feature. This is not the case for CriminalIntent. You call **`resolveActivity(…)`** to check for a working camera app, then gracefully disable the camera button if you do not find one.

Passing in `android:required="false"` handles this situation correctly. You tell Android that your app can work fine without the camera, but that some parts will be disabled as a result.

Challenge: Detail Display

While you can certainly see the image you display here, you cannot see it very well. For this first challenge, create a new **`DialogFragment`** that displays a zoomed-in version of your crime scene photo. When you press on the thumbnail, it should pull up the zoomed-in **`DialogFragment`**.

Challenge: Efficient Thumbnail Load

In this chapter, you had to use a crude estimate of the size you should scale down to. This is not ideal, but it works and is quick to implement.

With the out-of-the-box APIs, you can use a tool called **`ViewTreeObserver`**, an object that you can get from any view in your **`Activity`**'s hierarchy:

```
    ViewTreeObserver observer = mImageView.getViewTreeObserver();
```

You can register a variety of listeners on a **`ViewTreeObserver`**, including **`OnGlobalLayoutListener`**. This listener fires an event whenever a layout pass happens.

For this challenge, adjust your code so that it uses the dimensions of `mPhotoView` when they are valid and waits until a layout pass before initially calling **`updatePhotoView()`**.