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.

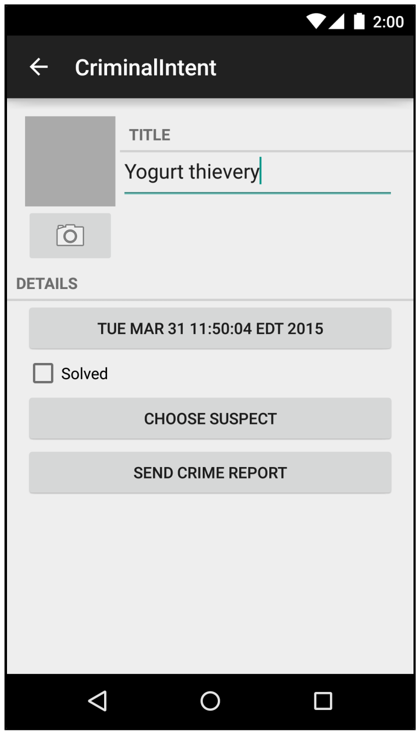
Taking a picture will involve a couple of new tools, used in combination with a tool you recently got to know: the implicit intent. An implicit intent can be used to start up the user's favorite camera application and receive a new picture from it.

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 user interface



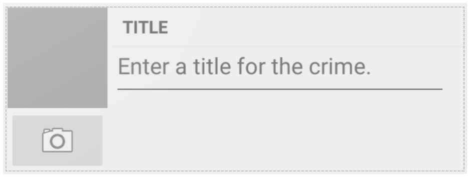
Dedicating an entire row to a thumbnail and a button would make your app look clunky and unprofessional. You do not want that, so you will arrange things nicely.

Including layout files

Your arrangement will include a large section that is the same in both layout and landscape versions of `fragment_crime.xml`. You can accomplish this by simply having this large section appear in both `res/layout/fragment_crime.xml` and `res/layout-land/fragment_crime.xml`. This is usually the right choice, but it is not the only choice. You can also use an *include*.

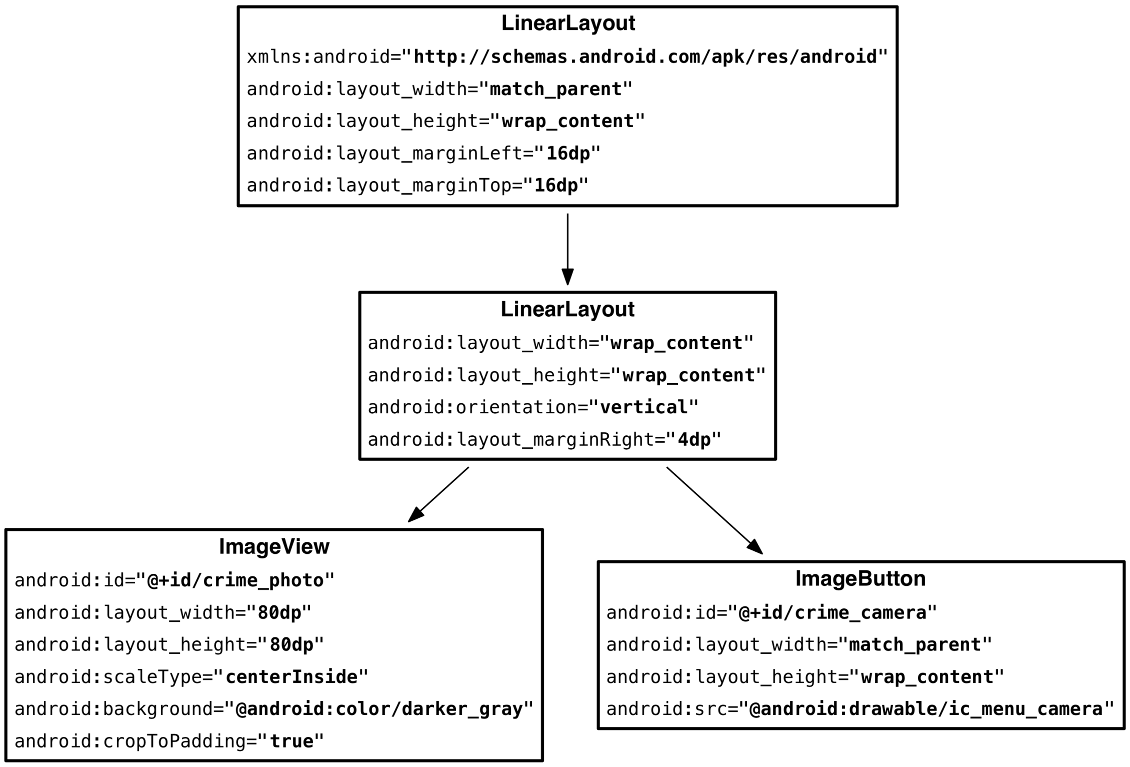
An *include* allows you to include one layout file in another. You are going to use one here for the common elements. The first step is to make a layout file that displays only the section of the view shown in Figure 16.2.

Figure 16.2 Camera and title



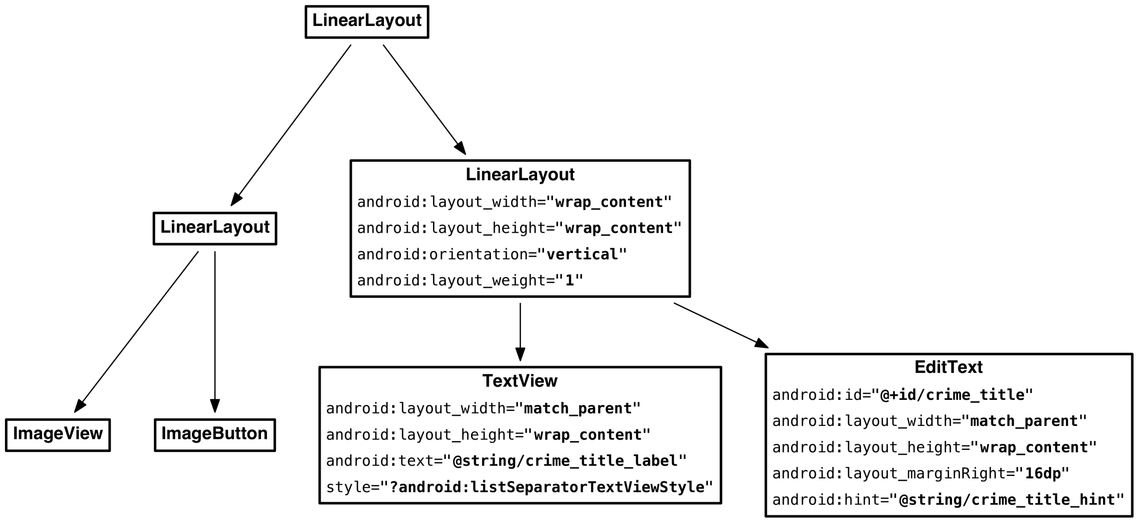
Now to make the layout file. Call it `view_camera_and_title.xml`. Build out the left-hand side first (Figure 16.3).

Figure 16.3 Camera view layout (`res/layout/view_camera_and_title.xml`)


LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:layout_marginLeft="16dp"
android:layout_marginTop="16dp"
LinearLayout
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:orientation="vertical"
android:layout_marginRight="4dp"
ImageView
android:id="@+id/crime_photo"
android:layout_width="80dp"
android:layout_height="80dp"
android:scaleType="centerInside"
android:background="@android:color/darker_gray"
android:cropToPadding="true"
ImageButton
android:id="@+id/crime_camera"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:src="@android:drawable/ic_menu_camera"

And then the right-hand side (Figure 16.4).

Figure 16.4 Title layout (`res/layout/view_camera_and_title.xml`)

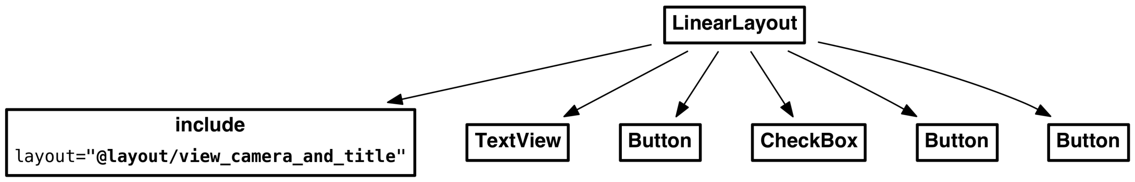

LinearLayout
LinearLayout
LinearLayout
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:orientation="vertical"
android:layout_weight="1"
ImageView
ImageButton
TextView
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:text="@string/crime_title_label"
style="?android:listSeparatorTextViewStyle"
EditText
android:id="@+id/crime_title"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:layout_marginRight="16dp"
android:hint="@string/crime_title_hint"

Use the design view to verify that your layout file looks like Figure 16.2.

Now you can use `include` tags to include this layout in your other layout files. When using the `include` tag, take note that the `layout` attribute does not use the normal `android` prefix.

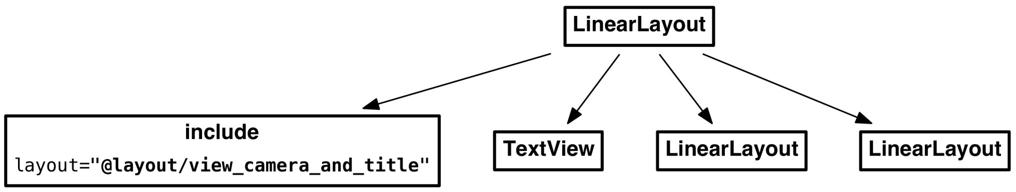
Modify your main layout file first (Figure 16.5).

Figure 16.5 Including camera layout (portrait) (`res/layout/fragment_crime.xml`)



And then your landscape layout (Figure 16.6).

Figure 16.6 Including camera layout (landscape) (`res/layout-land/fragment_crime.xml`)



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

Looks great, but to respond to presses on your **ImageButton** and to control the content of your **ImageView**, you need instance variables referring to each of them. No special steps are required to find views inside included layouts. Call **findViewById(int)** as usual on your inflated `fragment_crime.xml` and you will find views from `view_camera_and_title.xml`, too.

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

```
...
private CheckBox mSolvedCheckbox;
private Button mSuspectButton;
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 user interface for the time being. (You will wire those buttons up in a minute or two.)

External 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.

Normally, you would put them in 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).

Table 16.1 Basic file and directory methods in **`Context`**

Method	Purpose
`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.

If you are storing files that *only your current application* needs to use, these methods are exactly what you need.

On the other hand, if you need another application to write to those files, you are out of luck: while there is a `Context.MODE_WORLD_READABLE` flag you can pass in to **`openFileOutput(String, int)`**, it is deprecated, and not completely reliable in its effects on newer devices. If you are storing files to share with other apps or receiving files from other apps (files like stored pictures), you need to store them on *external storage* instead.

There are two kinds of external storage: primary, and everything else. All Android devices have at least one location for external storage: the primary location, which is located in the folder returned by **`Environment.getExternalStorageDirectory()`**. This may be an SD card, but nowadays it is more commonly integrated into the device itself. Some devices may have additional external storage. That would fall under "everything else."

`Context` provides quite a few methods for getting at external storage, too. These methods provide easy ways to get at your primary external storage, and kinda-sorta-easy ways to get at everything else. *All* of these methods store files in publicly available places, too, so be careful with them.

Table 16.2 External file and directory methods in **Context**

Method	Purpose
File getExternalCacheDir()	Returns a handle to a cache folder in primary external storage. Treat it like you do **getCacheDir()**, except a little more carefully. Android is even less likely to clean up this folder than the private storage one.
File[] getExternalCacheDirs()	Returns cache folders for multiple external storage locations.
File getExternalFilesDir(String)	Returns a handle to a folder on primary external storage in which to store regular files. If you pass in a type **String**, you can access a specific subfolder dedicated to a particular type of content. Type constants are defined in **Environment**, where they are prefixed with `DIRECTORY_`. For example, pictures go in `Environment.DIRECTORY_PICTURES`.
File[] getExternalFilesDirs(String)	Same as **getExternalFilesDir(String)**, but returns all possible file folders for the given type.
File[] getExternalMediaDirs()	Returns handles to all the external folders Android makes available for storing media – pictures, movies, and music. What makes this different from calling **getExternalFilesDir(Environment.DIRECTORY_PICTURES)** is that the media scanner automatically scans this folder. The media scanner makes files available to applications that play music, or browse movies and photos, so anything that you put in a folder returned by **getExternalMediaDirs()** will automatically appear in those apps.

Technically, the external folders provided above may not be available, since some devices use a removable SD card for external storage. In practice this is rarely an issue, because almost all modern devices have nonremovable internal storage for their “external” storage. So it is not worth going to extreme lengths to account for it. But we do recommended including simple code to guard against the possibility, which you will do in a moment.

Designating a picture location

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

Listing 16.2 Adding 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 does this.

Listing 16.3 Finding photo file location (`CrimeLab.java`)

```
public class CrimeLab {
    ...

    public Crime getCrime(UUID id) {
        ...
    }

    public File getPhotoFile(Crime crime) {
        File externalFilesDir = mContext
                .getExternalFilesDir(Environment.DIRECTORY_PICTURES);

        if (externalFilesDir == null) {
            return null;
        }

        return new File(externalFilesDir, crime.getPhotoFilename());
    }

    ...
```

This code does not create any files on the filesystem. It only returns **`File`** objects that point to the right locations. It does perform one check: it verifies that there is external storage to save them to. If there is no external storage, **`getExternalFilesDir(String)`** will return `null`. And so will this method.

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 16.4 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.

External storage permission

In general, you need a *permission* to write or read from external storage. Permissions are well-known string values you put in your manifest using the `<uses-permission>` tag. They tell Android that you want to do something that Android wants you to ask permission for.

Here, Android expects you to ask permission because it wants to enforce some accountability. You tell Android that you need to access external storage, and Android will then tell the user that this is one of the things your application does when they try to install it. That way, nobody is surprised when you start saving things to their SD card.

In Android 4.4, KitKat, they loosened this restriction. Since **`Context.getExternalFilesDir(String)`** returns a folder that is specific to your app, it makes sense that you would want to be able to read and write files that live there. So on Android 4.4 (API 19) and up, you do not need this permission for this folder. (But you still need it for other kinds of external storage.)

Add a line to your manifest that requests the permission to read external storage, but only up to API 18.

Listing 16.5 Requesting external storage permission (AndroidManifest.xml)

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

    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"
                     android:maxSdkVersion="18"
        />
    ...
```

The `maxSdkVersion` attribute makes it so that your app only asks for this permission on versions of Android that are older than API 19, Android KitKat.

Note that you are only asking to read external storage. There is also a `WRITE_EXTERNAL_STORAGE` permission, but you do not need it. You will not be writing anything to external storage: The camera app will do that for you.

Firing the intent

Now you are ready to fire the camera intent. The action you want is called `ACTION_CAPTURE_IMAGE`, 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_CAPTURE_IMAGE` 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 the whole thing 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`.

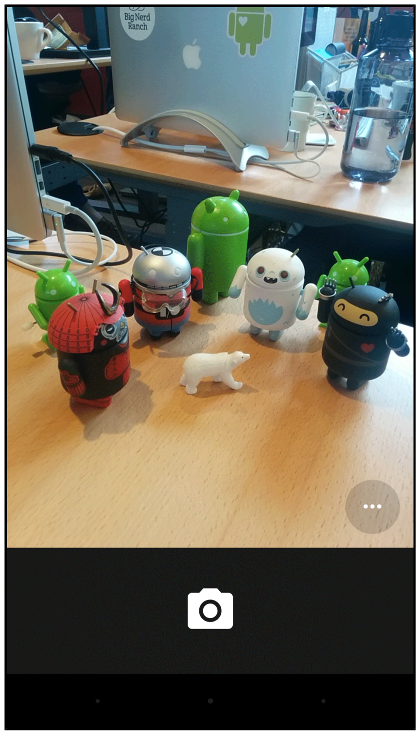
Write an implicit intent to ask for a new picture to be taken into the location saved in `mPhotoFile`. Add code to ensure that the button is disabled if there is no camera app, or if there is no location at which to save the photo. (To determine whether there is a camera app available, you will query **`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 16.6 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);

    if (canTakePhoto) {
        Uri uri = Uri.fromFile(mPhotoFile);
        captureImage.putExtra(MediaStore.EXTRA_OUTPUT, uri);
    }

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

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

    return v;
}
```

Run CriminalIntent and press the camera button to run your camera app (Figure 16.7).

Figure 16.7 [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 can do this by first scanning the file to see how big it is, next figuring out how much you need to scale it by to fit it in a given area, and finally rereading the file to create a scaled-down **Bitmap** object.

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

Listing 16.7 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) {
            if (srcWidth > srcHeight) {
                inSampleSize = Math.round(srcHeight / destHeight);
            } else {
                inSampleSize = Math.round(srcWidth / destWidth);
            }
        }

        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 on screen. 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 can wait until a layout pass happens, or you can 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 16.8 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 16.9 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 16.10 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) {
        updatePhotoView();
    }
}
```

Run 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 NFC, 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 16.11 Adding uses-feature tag (`AndroidManifest.xml`)

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.bignerdranch.android.criminalintent" >

    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"
                     android:maxSdkVersion="18"
        />
    <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 at all without that feature. This is not the case for CriminalIntent. You call **`resolveActivity(…)`** to check for a working camera app, then gracefully disable that 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.

For the More Curious: Using Includes

In this chapter, you used an include so that you would not have to repeat a large portion of your layout file across both landscape and portrait orientations. From that example, you can see how includes can be handy: they reduce typing and they help you follow the Don’t Repeat Yourself principle. But that does not mean that you should use them every time you have common layout items in landscape and portrait.

First, a detail about how includes work. In this chapter, you used an include tag without any `android` attributes. When you do that, the view you include gets all of the attributes it had in its original layout file.

However, you do not have to limit yourself to this. You can also add additional attributes. If you do, these attributes will be added directly to the root view you inflate. They will overwrite the original values of those attributes, too. This means that if you want to change the value of `layout_width`, you can.

The next thing we want to say about includes is cautionary. In this case, they have saved you some time and complexity, which is nice. But includes are not a perfect tool.

CriminalIntent’s views duplicate some **Button**s as well. You might be wondering why you did not get rid of that duplication with an include. The answer is: because this is not something we recommend.

One of the nice things about layout files is that they are authoritative: you can go to the layout file and see exactly how the view is supposed to be structured. Include files break this. You have to look at the layout file *and* all the files it includes to understand what is going on. This can quickly get irritating.

Visuals are often the part of an app that changes the most. When that is the case, perfectly following the DRY principle can mean that you end up worrying more about preserving your DRYitude than about actually building your interface. So try hard to be judicious, thoughtful, intentional, and restrained in how you apply includes in your view layer.

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**. **ViewTreeObserver** is 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()**.