Android 6.0's runtime permissions sound simple on the surface: just call checkSelfPermission() to see if you have the permission, then call requestPermissions() if you do not.

In practice, even a fairly simple app that uses these permissions has to add a remarkable amount of code, to handle all of the combinations of states, plus deal with some idiosyncrasies in the API. And, of course, since not everybody will be running a new device, we also have backwards compatibility to consider.

This standalone tutorial — not part of the EmPubLite series of tutorials throughout the rest of the core chapters — focuses on how to add the runtime permission support to an existing Android application.

As with the other code snippets in this book, if you are trying to copy and paste from the PDF itself, you will tend to have the best luck if you use the official Adobe Acrobat reader app.

Also, as part of working on this tutorial, you will be adding many snippets of Java code. You will need to add import statements for the new classes introduced by those code snippets. Just click on the class name, highlighted in red, in Android Studio and press <Alt>-<Enter> to invoke the quick-fix to add the required import statement.

**Step #0: Install the Android 6.0 SDK**

You are going to need the Android 6.0 (API 23) SDK Platform (or higher) in order to be able to implement runtime permission support. You may already have it, or you may need to install it.
If you open up Android Studio’s SDK Manager, via Tools > Android > “SDK Manager”, you may see Android 6.0 show up... or perhaps not:

![Image of Android Studio 1.3 SDK Manager, Sans Android 6.0](image)

*Figure 238: Android Studio 1.3 SDK Manager, Sans Android 6.0*

You may need to click the “Launch Standalone SDK Manager” link to bring up the classic SDK Manager, where you should see Android 6.0:
You will need the “SDK Platform” entry at minimum, and possibly an emulator “system image”.

If you have a device with Android 6.0+ on it, you are welcome to run the sample app, and it should allow you to take pictures and record videos. If you wish to run the sample app on an Android 6.0+ emulator, the permissions logic that we will be adding to the tutorial app will work, but it will not actually take pictures or record video. If your emulator image has 1+ cameras configured (see the “Advanced Settings” button when defining or editing your AVD), the activities to take a picture and record a video will come up but just show an indefinite progress indicator. If your emulator image has no cameras configured, those activities will just immediately finish and return control to our sample app’s main activity.

**Step #1: Import and Review the Starter Project**

Download the starter project ZIP archive and unzip it somewhere on your development machine.
Then, use File > New > Import Project to import this project into Android Studio. Android Studio may prompt you for additional updates from the SDK Manager (e.g., build tools), depending upon what you have set up on your development machine.

If you run the project on an Android 4.0+ device or emulator, you will see our highly-sophisticated user interface, consisting of two big buttons:

![Figure 240: Runtime Permissions Tutorial App, As Initially Written and Launched](image)

Tapping the “Take Picture” button will bring up a camera preview, with a floating action button (FAB) to take a picture:
TUTORIAL: RUNTIME PERMISSION SUPPORT

Tapping the FAB (and taking a picture) or pressing BACK will return you to the original two-button activity. There, tapping the “Record Video” button will bring up a similar activity, where you can press the green record FAB to start recording a video:
If you start recording, the FAB will change to a red stop button. Tapping that, or pressing BACK from either state, will return you to the initial two-button activity.

The application makes use of two third-party dependencies to pull all of this off:

- Philip Calvin's IconButton
- the author's CWAC-Cam2, which implements the photo and video activities

```
dependencies {
    compile 'com.commonsware.cwac:cam2:0.2.+'
    compile 'com.githang:com-phillipcalvin-iconbutton:1.0.1@aar'
}
```

Our two layouts, res/layout/main.xml and res/layout-land/main.xml, have two IconButton widgets in a LinearLayout, with equal weights so the buttons each take up half of the screen:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal">
```

MainActivity then uses CWAC-Cam2 to handle each of the button clicks:

```java
package com.commonsware.android.perm.tutorial;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.os.Environment;
import android.view.View;
import android.widget.Toast;
import com.commonsware.cwac.cam2.CameraActivity;
import com.commonsware.cwac.cam2.VideoRecorderActivity;
import java.io.File;

public class MainActivity extends Activity {

    private static final int RESULT_PICTURE_TAKEN = 1337;
    private static final int RESULT_VIDEO_RECORDED = 1338;
    private File rootDir;

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

        File downloads = Environment.getExternalStoragePublicDirectory(
                Environment.DIRECTORY_DOWNLOADS);

        rootDir = new File(downloads, "RuntimePermTutorial");
        rootDir.mkdirs();
    }

    @Override
```
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    Toast t = null;
    if (resultCode == RESULT_OK) {
        if (requestCode == RESULT_PICTURE_TAKEN) {
            t = Toast.makeText(this, R.string.msg_pic_taken, Toast.LENGTH_LONG);
        } else if (requestCode == RESULT_VIDEO_RECORDED) {
            t = Toast.makeText(this, R.string.msg_vid_recorded, Toast.LENGTH_LONG);
        }
        t.show();
    }
    public void takePicture(View v) {
        takePictureForRealz();
    }
    public void recordVideo(View v) {
        recordVideoForRealz();
    }
    private void takePictureForRealz() {
        Intent i = new CameraActivity.IntentBuilder(MainActivity.this)
            .to(new File(rootDir, "test.jpg"))
            .updateMediaStore()
            .build();
        startActivityForResult(i, RESULT_PICTURE_TAKEN);
    }
    private void recordVideoForRealz() {
        Intent i = new VideoRecorderActivity.IntentBuilder(MainActivity.this)
            .quality(VideoRecorderActivity.Quality.HIGH)
            .sizelimit(5000000)
            .to(new File(rootDir, "test.mp4"))
            .updateMediaStore()
            .forceClassic()
            .build();
        startActivityForResult(i, RESULT_VIDEO_RECORDED);
    }
}

The details of how CWAC-Cam2 works are not particularly relevant for the tutorial, but you can learn more about that later in the book if you are interested.

Taking pictures and recording videos require three permissions:

- CAMERA
- WRITE_EXTERNAL_STORAGE (where the output is going)
• RECORD_AUDIO (for videos)

Our manifest asks for none of these permissions:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest
    package="com.commonsware.android.perm.tutorial"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="1"
    android:versionName="1.0">
    <supports-screens
        android:anyDensity="true"
        android:largeScreens="true"
        android:normalScreens="true"
        android:smallScreens="true"
        android:xlargeScreens="true"/>
    <application
        android:allowBackup="false"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/Theme.Apptheme">
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

The permissions come from the CWAC-Cam2 library, courtesy of a process known as manifest merger.

You might wonder why we would bother doing this using a camera library in our own app. Most Android devices with camera hardware have a camera app, and most camera apps — particularly pre-installed camera apps — have activities that we could invoke to take pictures or record videos. However, these activities are infrequently tested, and many do not work properly. Since they are unreliable, you may be happier using something that is a library, packaged in your app.

Note that MainActivity has some seemingly superfluous bits of code:

• We find the two buttons in the inflated layout and assign them to takePicture and recordVideo fields... but then never use them
We delegate the actual CWAC-Cam2 work to `takePictureForRealz()` and `recordVideoForRealz()`... instead of just doing that work in the `takePicture()` and `recordVideo()` methods invoked by the buttons.

The reason for those apparent inefficiencies is to reduce the amount of work it will take you to add the runtime permissions, by handling a tiny bit of bookkeeping ahead of time.

**Step #2: Update Gradle for Android 6.0**

By default, if you run this app from your IDE on an Android 6.0 device, nothing appears to be different. The app runs as it did.

If you were to install it via a download, such as from a Web site, the installation process looks as it does on earlier Android versions, prompting the user for each of the permissions:

![Figure 243: Installing the Tutorial App From the Web](image)

However, the user can still go into Settings and elect to disable our access to those permissions:
In our case, not all those permissions are always needed, and it would be useful to know whether or not we hold a permission, and so adopting the new runtime permission model would seem to be a good idea.

The first step on the road to doing that is to adjust some values in our app/ module’s build.gradle file:

- Change compileSdkVersion to 23, as we need to use methods from the latest SDK
- Change buildToolsVersion to 23.0.0, to keep it in sync with the compileSdkVersion, and
- Change targetSdkVersion to 23, to tell Android that our app was written with the runtime permission model in mind

This will give you an android closure like:

```java
android {
    compileSdkVersion 23
    buildToolsVersion "23.0.0"

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 23
    }
}
```
Step #3: Review the Planned UX

So, our app is here to take pictures and record videos. Of the three permissions that our app is requesting in total, two are essential for the app to do anything meaningful: CAMERA and WRITE_EXTERNAL_STORAGE. RECORD_AUDIO, by contrast, is not needed if the user only wants to take pictures.

Part of the objective of the runtime permissions system is to allow you to lazy-request permissions that many users may not need. If there is some fringe feature in your app that, say, needs READ_CONTACTS, rather than force everyone to give you READ_CONTACTS, you can request it only of users who go down the path in your UI that leads to the feature that needs READ_CONTACTS-secured capabilities.

Hence, we will only ask for the RECORD_AUDIO permission if the user taps the “Record Video” button.

For the other two permissions, we could take the approach of asking for them only when the user taps either of the two buttons. However, those permissions are essential for app operation, and so another approach is to ask for those permissions on first run of the app, and only worry about them on button clicks if our original request was rejected. You might have some sort of “onboarding” welcome tutorial that explains a bit why we are going to ask for the permissions. Or, you could just ask for the permissions and hope that users will have seen those sorts of request dialogs before, as this app will do (for simplicity as much as anything else).

When the user clicks a button, we need to double-check to see if we have the permissions, and perhaps ask the user again for those permissions. Along the way, we may wish to show some “rationale” — an explanation, in our own UI, of why we need the permissions that we asked for previously and the user said “no”.

If, however, the user not only declines to grant us some permission, but also checks the checkbox indicating that we are not to keep asking, we may as well disable the affected button(s), as the user cannot use that functionality. Alternatively, we might keep the buttons clickable, but instead of doing the actual work (which we cannot do due to lack of permissions), show a message directing the user to the Settings app to flip the switches and grant the permissions to our app. The app in this tutorial will settle for just disabling the buttons.
So, for each of our permissions, we are in one of four states:

1. We have never asked for the permission before
2. We asked for the permission, and the user granted it
3. We asked for the permission, and either the user rejected our request, or perhaps granted it but then changed their mind and turned the permission back off in Settings
4. We asked for the permission, and not only did the user reject it, but the user also indicated (via a checkbox) that we are not to ask again

We are going to need to distinguish between these four states as part of our app logic, in order to present the proper behavior in each case.

**Step #4: Detect the First Run**

If we are going to ask for the CAMERA and WRITE_EXTERNAL_STORAGE permissions on the first run of our app, we need to know when the first run of our app has happened. To do this, we will take a typical approach, using a boolean value in SharedPreferences to determine if we have run before.

With that in mind, add the following constant declaration to MainActivity:

```java
private static final String PREF_IS_FIRST_RUN = "firstRun";
```

This will serve as the key to our boolean SharedPreferences value.

Then, add the following prefs data member to MainActivity:

```java
private SharedPreferences prefs;
```

Next, initialize prefs in MainActivity, shortly after the setContentView() call:

```java
prefs = PreferenceManager.getDefaultSharedPreferences(this);
```

Then, add the following method to MainActivity:

```java
private boolean isFirstRun() {
    boolean result = prefs.getBoolean(PREF_IS_FIRST_RUN, true);

    if (result) {
        prefs.edit().putBoolean(PREF_IS_FIRST_RUN, false).apply();
    }
```
This retrieves the existing value, defaulting to true if there is no such value. If we get that default back, we then update the SharedPreferences to save false for future use.

Finally, at the bottom of onCreate() of MainActivity, add the following lines:

```java
if (isFirstRun()) {
    // TODO
}
```

We will replace that comment shortly.

**Step #5: On First Run, Ask For Permissions**

As was covered back in Step #3, we want to ask for the CAMERA and WRITE_EXTERNAL_STORAGE permissions on the first run of our app. To do that, we need to call requestPermissions() from within that if block we added in the previous step.

`requestPermissions()` takes two parameters:

1. A String array of the fully-qualified names of the permissions that we want
2. An int that will be returned to us in an onRequestPermissionsResult() callback method, so we can distinguish the results of one `requestPermissions()` call from another

You might wonder why, when adding this in 2015, the Android engineers did not use some sort of a callback object, rather than mess around with int values. Sometimes, the author of this book wonders too.

But, regardless, that is what we need, and we had best start implementing it.

First, to make our code a bit easier to read, add the following static import statements to MainActivity:

```java
import static android.Manifest.permission.CAMERA;
import static android.Manifest.permission.RECORD_AUDIO;
import static android.Manifest.permission.WRITE_EXTERNAL_STORAGE;
```
If you have not seen this Java syntax before, a static import basically imports a static method or field from a class (in this case, from Manifest.permission). The result of the import is that we can refer to the imported items as if they were static items on our own class. So, we can just have a reference to CAMERA, for example, rather than having to spell out something like Manifest.permission.CAMERA every time.

Next, add the following static String array to MainActivity, one that uses some of our newly-added static imports:

```java
private static final String[] PERMS_TAKE_PICTURE = {
    CAMERA,
    WRITE_EXTERNAL_STORAGE
};
```

Also add the following int constant to MainActivity:

```java
private static final int RESULT_PERMS_INITIAL = 1339;
```

Now, we can request our permissions. However, if we call requestPermissions() on Activity, we have a problem: that method was only added in API Level 23. If our minSdkVersion were 23 or higher, that would not be a problem. However, our minSdkVersion is 15, and it would be nice to support Android 4.x and 5.x devices.

The recommended solution for this is to use an ActivityCompat class, supplied by the support-v4 portion of the Android Support library. This class contains, among other things, a requestPermissions() static method that will confirm that we are on a device new enough to support requestPermissions(). On older devices, it gracefully degrades (in this case, doing nothing, as we already have our permissions).

So, with that in mind, edit the build.gradle file in the app/ module to add in support-v4:

```gradle
apply plugin: 'com.android.application'

repositories {
    maven {
        url "https://repo.commonsware.com.s3.amazonaws.com"
    }
}

dependencies {
    compile 'com.commonsware.cwac:cam2:0.2.+'
    compile 'com.githang:com-phillipcalvin-iconbutton:1.0.1@aar'
    compile 'com.android.support:support-v4:23.0.1'
}
```
android {
    compileSdkVersion 23
    buildToolsVersion "23.0.0"

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 23
    }
}

Then, update the if block in onCreate() of MainActivity to look like:

```java
if (isFirstRun()) {
    ActivityCompat.requestPermissions(this, PERMS_TAKE_PICTURE,
                                       RESULT_PERMS_INITIAL);
}
```

The corresponding callback for requestPermissions() is
onRequestPermissionsResult(). So, add a stub implementation of this callback to
MainActivity:

```java
@override
public void onRequestPermissionsResult(int requestCode,
                                        String[] permissions,
                                        int[] grantResults) {
  // TODO
}
```

As before, we will be replacing that // TODO a bit later in the tutorial.

At this point, run the app on your Android 6.0 environment. Immediately, you
should be prompted for the permissions:
Figure 245: Tutorial App, Showing CAMERA Permission Request

Figure 246: Tutorial App, Showing WRITE_EXTERNAL_STORAGE Permission Request
Then, uninstall the app. That way, no matter whether you accepted or declined those permissions, the next time you run the app, you are “starting from a clean slate”.

**Step #6: Check for Permissions Before Taking a Picture**

If we are lucky, our users will grant us the permissions that we requested. We will not always be lucky; some users will reject our request. Furthermore, some users might change these permissions for our app in Settings, granting or revoking them as those users see fit.

So, when the user taps the “Take Picture” button, we need to double-check to see if we actually have the permissions that we need. If we do not, we cannot go ahead and take the picture “for realz”, as we will crash with a SecurityException, because we lack the permission.

With that in mind, add the following method `hasPermission()` method to `MainActivity`:

```java
private boolean hasPermission(String perm) {
    return ContextCompat.checkSelfPermission(this, perm) == PackageManager.PERMISSION_GRANTED;
}
```

This is just a convenience method to reduce clutter elsewhere in the class when we try to determine whether or not we have a permission. This method uses `ContextCompat`, another compatibility class from support-v4, to see if we have the supplied permission. While we could call `checkSelfPermission()` directly on our `MainActivity`, we would run into the same problem that we did with `requestPermissions()` — `checkSelfPermission()` only exists on API Level 23+. The `ContextCompat` edition of the method gracefully degrades on older devices, returning true, since we already have the permission.

Next, add the following `canTakePicture()` method to `MainActivity`:

```java
private boolean canTakePicture() {
    return hasPermission(CAMERA) && hasPermission(WRITE_EXTERNAL_STORAGE));
}
```

Here, `canTakePicture()` simply checks to see if we can take a picture, by checking whether we have the `CAMERA` and `WRITE_EXTERNAL_STORAGE` permissions.
Then, modify the takePicture() method of MainActivity to look like this:

```java
public void takePicture(View v) {
    if (canTakePicture()) {
        takePictureForRealz();
    }
}
```

Here, we only try taking the picture if we have the permissions.

Of course, if we do not have the permissions, right now we are ignoring the user clicks on our “Take Picture” button. We really should offer more feedback here, and we will be tackling that little problem in later steps of this tutorial.

Now, run the app on an Android 6.0 environment. When Android prompts you for the permissions, accept them. Then, tap the “Take Picture” button, and you should be able to take a picture.

Then, uninstall the app and run it again, this time rejecting the permissions when asked. Then, tap the “Take Picture” button, and you should get no response from the app.

Finally, uninstall the app.

**Step #7: Detect If We Should Show Some Rationale**

Having no response to tapping the “Take Picture” button, when we do not have the requisite permissions, is not a very good user experience. We should ask again for those permissions... if there is a chance that the user will actually grant them to us.

That chance will be improved if we explain to them, a bit more, why we keep asking for these permissions. Android 6.0 has a shouldShowRequestPermissionRationale() that we can use to decide whether we should show some UI (and then later ask for the permissions again) or whether the user has checked the “don’t ask again” checkbox and we should leave them alone.

With that in mind, add the following method to MainActivity:

```java
private boolean shouldShowTakePictureRationale() {
    return ActivityCompat.shouldShowRequestPermissionRationale(
            this, CAMERA) ||
            ActivityCompat.shouldShowRequestPermissionRationale(this,
                    WRITE_EXTERNAL_STORAGE);
}
```
This `shouldShowTakePictureRationale()` simply checks to see if we need to show rationale for any of the permissions required to take a picture. It uses the `shouldShowRequestPermissionRationale()` method, which will return `false` if:

- You have never asked for the permission, or
- You have asked for the permission a few times, and the last time out, the user not only denied the permission, but also checked the checkbox to prevent you from asking for permission again in the future

Otherwise, `shouldShowRequestPermissionRationale()` will return `true`.

As with the other runtime permission-specific methods used here in `MainActivity`, while there is one in the SDK for direct use (`shouldShowRequestPermissionRationale()` on `Activity`), it was added in API Level 23. The backport (`shouldShowRequestPermissionRationale()` on `ActivityCompat`) will handle cases where we are running on an older version of Android.

Then, modify the existing `takePicture()` method to look like this:

```java
public void takePicture(View v) {
    if (canTakePicture()) {
        takePictureForRealz();
    } else if (shouldShowTakePictureRationale()) {
        // TODO
    }
}
```

So, now we are checking to see if we should show the user an explanation for the permissions... though we are not doing that just yet. We will get to that in the next step.

**Step #8: Add a Rationale UI and Re-Request Permissions**

We need to do something to explain to the user why we need these permissions.

A poor choice would be to display a `Toast`. Those are time-limited and so are not good for showing longer messages.
We might display a dialog or a snackbar... but we have not talked about how to do either of those just yet in this book.

We might display something from our help system, or go through the introductory tutorial again, or something like that... but this app does not have any of those things.

So, we will instead take a very crude UI approach: adding a hidden panel with our message that we will show when needed. Since this is not nearly as refined as a Toast, we will call this panel the breadcrumb.

With that as background, let’s add a TextView to our res/layout/main.xml file that is the breadcrumb itself:

```xml
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextView
        android:id="@+id/breadcrust"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:background="@color/accent"
        android:gravity="center"
        android:padding="8dp"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:visibility="gone"/>

    <com.phillipcalvin.iconbutton.IconButton
        android:id="@+id/take_picture"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_margin="4dp"
        android:layout_weight="1"
        android:drawableRight="@drawable/ic_camera_black_48dp"
        android:onClick="takePicture"
        android:text="Take Picture"
        android:textAppearance="?android:attr/textAppearanceLarge"
        app:iconPadding="16dp"/>

    <com.phillipcalvin.iconbutton.IconButton
        android:id="@+id/record_video"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_margin="4dp"
        android:layout_weight="1"
        android:drawableRight="@drawable/ic_videocam_black_48dp"/>

</LinearLayout>
```
Here, we are having it take up its share of the space, the same as the two buttons (android:layout_weight="1") and giving it a yellow background (android:background="@color/accent"). The android:textAppearance="?android:attr/textAppearanceLarge" is Android’s cumbersome way of saying “use the standard large-type font”. Finally, android:visibility="gone" means that this TextView actually will not be seen, until we make it visible ourselves in Java code.

We need to add a similar TextView to the res/layout-land/main.xml file, simply inverting the axes for the width, height, and weight:

```xml
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal">

    <TextView
        android:id="@+id/breadcrust"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1"
        android:background="@color/accent"
        android:gravity="center"
        android:padding="8dp"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:visibility="gone"/>

    <com.phillipcalvin.iconbutton.IconButton
        android:id="@+id/take_picture"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_margin="4dp"
        android:layout_weight="1"
        android:drawableRight="@drawable/ic_camera_black_48dp"
        android:onClick="takePicture"
        android:text="Take Picture"
        android:textAppearance="?android:attr/textAppearanceLarge"
        app:iconPadding="16dp"/>

    <com.phillipcalvin.iconbutton.IconButton
        android:id="@+id/record_video"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_margin="4dp"
```
Next, add a data member for the breadcrumb to MainActivity:

```java
private TextView breadcrumb;
```

Then, in `onCreate()` of `MainActivity`, add a call to `findViewById()` to look up the breadcrumb:

```java
breadcrumb=(TextView) findViewById(R.id.breadcrust);
```

Next, in `res/values/strings.xml`, add in a string resource for the message we want to show in the breadcrumb when we are going to ask the user (again) for permission to take pictures:

```xml
<string name="msg_take_picture">You need to grant us permission! Tap the Take Picture button again, and we will ask for permission.</string>
```

So, what we want to have happen when the user taps the “Take Picture” button is:

- If we have permission to take the picture, take the picture
- If we do not have permission, but the user can see the breadcrumb (and so can see our rationale for requesting the permission), request the permissions again
- If we do not have permission, and the breadcrumb is not visible, then we need to show the breadcrumb with our rationale message

To that end, modify `takePicture()` on `MainActivity` to look like this:

```java
public void takePicture(View v) {
    if (canTakePicture()) {
        takePictureForRealz();
    } else if (breadcrumb.getVisibility()==View.GONE && shouldShowTakePictureRationale()) {
        breadcrumb.setText(R.string.msg_take_picture);
        breadcrumb.setVisibility(View.VISIBLE);
    } else {
        breadcrumb.setVisibility(View.GONE);
        ActivityCompat.requestPermissions(this, netPermissions(PERMS_TAKE_PICTURE), RESULT_PERMS_TAKE_PICTURE);
    }
}
```
If breadcrumb is visible, we make it GONE again and call requestPermissions. If breadcrumb is not visible, we make it VISIBLE and set its message to the string resource that we defined.

There are some things missing. The biggest one is the netPermissions() method:

```java
private String[] netPermissions(String[] wanted) {
    ArrayList<String> result = new ArrayList<String>();
    for (String perm : wanted) {
        if (!hasPermission(perm)) {
            result.add(perm);
        }
    }
    return result.toArray(new String[result.size()]);
}
```

This method iterates over our input string array of permissions and filters out those that we already hold. This is needed because a call to requestPermissions() requests *every* permission that we ask for... even permissions that the user has already granted. For example, suppose that on the initial run of our app, the user granted the WRITE_EXTERNAL_STORAGE permission but declined to grant the CAMERA permission. We only want to ask the user for the CAMERA permission. Ideally, requestPermissions() would look at our array and filter out those permissions that we were already granted, asking the user for the remainder. Unfortunately, requestPermissions() does not do that, so we have to do the filtering ourselves, as we are in netPermissions().

netPermissions() just iterates over the array of permission names and uses a hasPermission() method to filter out ones that we already hold.

Also, your IDE should complain that RESULT_PERMS_TAKE_PICTURE is not defined, so add that as another constant on MainActivity:

```java
private static final int RESULT_PERMS_TAKE_PICTURE=1340;
```

If we call requestPermissions() and the user grants the permissions, we should go ahead and take the picture. We also need to deal with the case where the user has denied the permission and checked the “stop asking” checkbox, as our requestPermissions() call will route straight to onRequestPermissionsResult() without prompting the user. So, we need to add some more logic to the
onRequestPermissionsResult() callback method in MainActivity, so alter yours to look like this:

```java
@Override
public void onRequestPermissionsResult(int requestCode, String[] permissions,
                                        int[] grantResults) {
    boolean sadTrombone = false;
    if (requestCode == RESULT_PERMS_TAKE_PICTURE) {
        if (canTakePicture()) {
            takePictureForRealz();
        } else if (!shouldShowTakePictureRationale()) {
            sadTrombone = true;
        }
    }
    if (sadTrombone) {
        Toast.makeText(this, R.string.msg_no_perm,
                        Toast.LENGTH_LONG).show();
    }
}
```

Here, if the requestCode is the one we used in our call to requestPermissions() (RESULT_PERMS_TAKE_PICTURE), and if we have permission now to take a picture, we take the picture.

If, on the other hand, we are in onRequestPermissionsResult() (so we know we have asked for the permission) and shouldShowTakePictureRationale() returns false, we know that we cannot possibly get the permission anymore, other than by the user going into the Settings app and manually granting it to us. So, we show a Toast to inform the user about this.

Your IDE will complain that there is no R.string.msg_no_perm value, so add another string resource to your strings.xml file:

```
<string name="msg_no_perm">Sorry, you did not give us permission!</string>
```

Now, run the app on your Android 6.0 environment. When the app asks for permissions on the first run, reject at least one of them. Then, tap the “Take Picture” button, and you should see the breadcrust appear:
If you tap the “Take Picture” button again, the breadcrupt will go away, and you will be prompted for any permissions you did not grant previously. If you reject any permissions here, you are back where you were; if you accept all permissions, the app will allow you to take a picture. If, instead, you deny all permissions and check the “do not ask again” checkbox for at least one of them, you should see the Toast appear when you try tapping “Take Picture”.

Then, uninstall the app.

**Step #9: Check for Permissions Before Recording a Video**

So far, we have ignored the “Record Video” button, so let’s start wiring up support for it as well. The big difference with this button — besides recording a video instead of taking a picture — is that we are not asking for the RECORD_AUDIO permission up front.

However, that does not change some of the basics, like seeing if we have permission to record videos and only trying to record videos if we do.
First, add the following method to MainActivity:

```java
private boolean canRecordVideo() {
    return canTakePicture() && hasPermission(RECORD_AUDIO);
}
```

This canRecordVideo() method will return true if we can take a picture and have the RECORD_AUDIO permission. canTakePicture() already checks the CAMERA and WRITE_EXTERNAL_STORAGE permissions, so we are just chaining on the additional permission check.

Then, modify recordVideo() in MainActivity to use this:

```java
public void recordVideo(View v) {
    if (canRecordVideo()) {
        recordVideoForRealz();
    }
}
```

If you run the sample app, and you tap the “Record Video” button, you should get no response, as we have never asked for the RECORD_AUDIO permission, so canRecordVideo() should return false. Then, uninstall the app.

**Step #10: Detect If We Should Show Some Rationale (Again)**

We also need to arrange to show the breadcrumb, with a video-related message, if we do not have permission to take a video but could get it.

So, add the following method to MainActivity:

```java
private boolean shouldShowRecordVideoRationale() {
    return shouldShowTakePictureRationale() ||
           ActivityCompat.shouldShowRequestPermissionRationale(this,
                       RECORD_AUDIO);
}
```

Once again, we are checking to see if we need to show a rationale either because of camera-related permissions (shouldShowTakePictureRationale()) or because of the RECORD_AUDIO permission.

Then, add a couple of additional branches to the recordVideo() method:
recordVideoForRealz();
else if (breadcrust.getVisibility()==View.GONE
      && shouldShowRecordVideoRationale()) {
    breadcrust.setText(R.string.msg_record_video);
    breadcrust.setVisibility(View.VISIBLE);
} else {
    breadcrust.setVisibility(View.GONE);
    ActivityCompat.requestPermissions(this,
        netPermissions(PERMS_ALL), RESULT_PERMS_RECORD_VIDEO);
}

Your IDE will complain that you are missing two constants. One is PERMS_ALL, the list of permissions needed to record a video, so add that to MainActivity:

private static final String[] PERMS_ALL={
    CAMERA,
    WRITE_EXTERNAL_STORAGE,
    RECORD_AUDIO
};

Also, we need to add RESULT_PERMS_RECORD_VIDEO to MainActivity:

private static final int RESULT_PERMS_RECORD_VIDEO=1341;

This is the same flow as we had with the takePicture() method:

- If we have permission to record the video, go ahead and do so
- If we do not, and we are not showing the breadcrust, but we should show some rationale, populate and show the breadcrust
- Otherwise, make sure the breadcrust is GONE and request our permissions

Finally, modify onRequestPermissionsResult() in MainActivity to record the video if we now have permission to do so, by adding the else if block:

@Override
public void onRequestPermissionsResult(int requestCode,
                                             String[] permissions,
                                             int[] grantResults) {
    boolean sadTrombone=false;
    if (requestCode==RESULT_PERMS_TAKE_PICTURE) {
        if (canTakePicture()) {
            takePictureForRealz();
        } else if (!shouldShowTakePictureRationale()) {
            sadTrombone=true;
        }
    }
}
else if (requestCode==RESULT_PERMS_RECORD_VIDEO) {
    if (canRecordVideo()) {
        recordVideoForRealz();
    } else if (!shouldShowRecordVideoRationale()) {
        sadTrombone=true;
    }
    if (sadTrombone) {
        Toast.makeText(this, R.string.msg_no_perm,
                     Toast.LENGTH_LONG).show();
    }
}

If you run the app and tap the “Record Video” button, you should be asked for all required permissions right away, as we have never asked you for RECORD_AUDIO. If you decline one or more of the permissions, and tap “Record Video” a second time, the breadcrust should appear. If you tap “Record Video” a third time, the breadcrust should vanish and you should be prompted for the permissions again. And, if you deny all permissions while checking the checkbox, you should see the Toast telling you that we cannot record a video. Then, uninstall the app.

Step #11: Support Configuration Changes

The final thing that we need to do is take configuration changes into account.

Specifically, we need to track whether the breadcrust is visible, and if so, what message is displayed. That way, when our activity is destroyed and recreated on a configuration change, we can restore the breadcrust to its last state as well.

Add the following constant to MainActivity:

```java
private static final String STATE_BREADCRUST=
    "com.commonsware.android.perm.tutorial.breadcrust";
```

We will use STATE_BREADCRUST as the key to the Bundle value that we will store in the saved instance state.

Then, add onSaveInstanceState() and onRestoreInstanceState() methods to MainActivity:

```java
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    if (breadcrust.getVisibility()==View.VISIBLE) {
```

519
If the breadcrumb is visible, we save the message from the breadcrumb in the Bundle. In `onRestoreInstanceState()`, we make the breadcrumb be visible if we have a message, where we also put that message into the breadcrumb.

**NOTE:** This is a sloppy approach that works only because this app only supports one language. Otherwise, in case of a locale change, we would be saving the message in the old language in the Bundle and reapplying it, while the rest of our UI is in the new language. A better implementation would track which of the two messages we need (e.g., via int string resource IDs) so we can reapply the resources, pulling in the proper translations. That requires a bit more bookkeeping, and this sample is already annoyingly long. However, just bear in mind that how we are saving the state here is crude and only effective for this limited scenario.

If you run the app one last time and get the breadcrumb to appear, rotating the device or otherwise triggering a configuration change will not lose the breadcrumb, even though our activity will be destroyed and recreated along the way.

At this point, your `MainActivity` should resemble the following:

```java
package com.commonsware.android.perm.tutorial;

import android.app.Activity;
import android.content.Intent;
import android.content.SharedPreferences;
import android.content.pm.PackageManager;
import android.os.Bundle;
import android.os.Environment;
import android.preference.PreferenceManager;
import android.support.v4.app.ActivityCompat;
import android.support.v4.content.ContextCompat;
import android.view.View;
import android.widget.TextView;
import android.widget.Toast;
```
import com.commonsware.cwac.cam2.CameraActivity;
import com.commonsware.cwac.cam2.VideoRecorderActivity;
import java.io.File;
import java.util.ArrayList;
import static android.Manifest.permission.CAMERA;
import static android.Manifest.permission.RECORD_AUDIO;
import static android.Manifest.permission.WRITE_EXTERNAL_STORAGE;

public class MainActivity extends Activity {
    private static final String[] PERMS_ALL = {
        CAMERA,
        WRITE_EXTERNAL_STORAGE,
        RECORD_AUDIO
    };
    private static final String[] PERMS_TAKE_PICTURE = {
        CAMERA,
        WRITE_EXTERNAL_STORAGE
    };
    private static final int RESULT_PICTURE_TAKEN = 1337;
    private static final int RESULT_VIDEO_RECORDERED = 1338;
    private static final int RESULT_PERMS_INITIAL = 1339;
    private static final int RESULT_PERMS_TAKE_PICTURE = 1340;
    private static final int RESULT_PERMS_RECORD_VIDEO = 1341;
    private static final String PREF_IS_FIRST_RUN = "firstRun";
    private static final String STATE_BREADCRUST = "com.commonsware.android.perm.tutorial.breadcrust";

    private File rootDir;
    private SharedPreferences prefs;
    private TextView breadcrumb;

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

        prefs = PreferenceManager.getDefaultSharedPreferences(this);
        breadcrumb = (TextView) findViewById(R.id.breadcrust);

        File downloads = Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS);
        rootDir = new File(downloads, "RuntimePermTutorial");
        rootDir.mkdirs();

        if (isFirstRun()) {
            ActivityCompat.requestPermissions(this, PERMS_TAKE_PICTURE, RESULT_PERMS_INITIAL);
        }
    }

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

        if (breadcrumb.getVisibility() == View.VISIBLE) {
            outState.putCharSequence(STATE_BREADCRUST, breadcrumb.getText());
        }
    }
}
@Override
protected void onRestoreInstanceState(Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);
    CharSequence cs = savedInstanceState.getCharSequence(STATE_BREADCRUST);
    if (cs != null) {
        breadcrumb.setVisibility(View.VISIBLE);
        breadcrumb.setText(cs);
    }
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    Toast t = null;
    if (resultCode == RESULT_OK) {
        if (requestCode == RESULT_PICTURE_TAKEN) {
            t = Toast.makeText(this, R.string.msg_pic_taken, Toast.LENGTH_LONG);
        } else if (requestCode == RESULT_VIDEO_RECORD) {
            t = Toast.makeText(this, R.string.msg_vid_recorded, Toast.LENGTH_LONG);
        }
    }
    t.show();
}

public void takePicture(View v) {
    if (canTakePicture()) {
        takePictureForRealz();
    } else if (breadcrumb.getVisibility() == View.GONE && shouldShowTakePictureRationale()) {
        breadcrumb.setText(R.string.msg_take_picture);
        breadcrumb.setVisibility(View.VISIBLE);
    } else {
        breadcrumb.setVisibility(View.GONE);
        ActivityCompat.requestPermissions(this, netPermissions(PERMS_TAKE_PICTURE), RESULT_PERMS_TAKE_PICTURE);
    }
}

@Override
public void onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults) {
    boolean sadTrombone = false;
    if (requestCode == RESULT_PERMS_TAKE_PICTURE) {
        if (canTakePicture()) {
            takePictureForRealz();
        }
    }
}
else if (!shouldShowTakePictureRationale()) {
    sadTrombone = true;
}
else if (requestCode==RESULT_PERMS_RECORD_VIDEO) {
    if (canRecordVideo()) {
        recordVideoForRealz();
    } else if (!shouldShowRecordVideoRationale()) {
        sadTrombone = true;
    }
}
if (sadTrombone) {
    Toast.makeText(this, R.string.msg_no_perm, Toast.LENGTH_LONG).show();
}

public void recordVideo(View v) {
    if (canRecordVideo()) {
        recordVideoForRealz();
    } else if (breadcrust.getVisibility()==View.GONE && shouldShowRecordVideoRationale()) {
        breadcrust.setText(R.string.msg_record_video);
        breadcrust.setVisibility(View.VISIBLE);
    } else {
        breadcrust.setVisibility(View.GONE);
        ActivityCompat.requestPermissions(this, netPermissions(PERMS_ALL), RESULT_PERMS_RECORD_VIDEO);
    }
}

private void takePictureForRealz() {
    Intent i = new CameraActivity.IntentBuilder(MainActivity.this)
        .to(new File(rootDir, "test.jpg")
        .updateMediaStore()
        .build();

    startActivityForResult(i, RESULT_PICTURE_TAKEN);
}

private void recordVideoForRealz() {
    Intent i = new VideoRecorderActivity.IntentBuilder(MainActivity.this)
        .quality(VideoRecorderActivity.Quality.HIGH)
        .sizeLimit(5000000)
        .to(new File(rootDir, "test.mp4")
        .updateMediaStore()
        .forceClassic()
        .build();

    startActivityForResult(i, RESULT_VIDEO_RECORDED);
}

private boolean isFirstRun() {

boolean result = prefs.getBoolean(PREF_IS_FIRST_RUN, true);

if (result) {
    prefs.edit().putBoolean(PREF_IS_FIRST_RUN, false).apply();
}

return result;

private boolean hasPermission(String perm) {
    return ContextCompat.checkSelfPermission(this, perm) == PackageManager.PERMISSION_GRANTED;
}

private boolean canTakePicture() {
    return hasPermission(CAMERA) && hasPermission(WRITE_EXTERNAL_STORAGE));
}

private boolean shouldShowTakePictureRationale() {
    return ActivityCompat.shouldShowRequestPermissionRationale(this, CAMERA) ||
           ActivityCompat.shouldShowRequestPermissionRationale(this, WRITE_EXTERNAL_STORAGE));
}

private String[] netPermissions(String[] wanted) {
    ArrayList<String> result = new ArrayList<String>();

    for (String perm : wanted) {
        if (!hasPermission(perm)) {
            result.add(perm);
        }
    }

    return result.toArray(new String[result.size()]);
}

private boolean canRecordVideo() {
    return canTakePicture() && hasPermission(RECORD_AUDIO));
}

private boolean shouldShowRecordVideoRationale() {
    return ActivityCompat.shouldShowRequestPermissionRationale(shouldShowTakePictureRationale() ||
           ActivityCompat.shouldShowRequestPermissionRationale(this, RECORD_AUDIO));
}

And your tutorial is now complete.