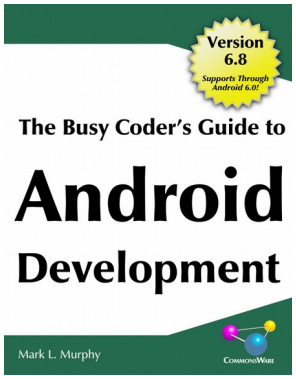
Android 6.0 Runtime Permissions

a CommonsWare Code Lab



Welcome to the Android 6.0 Runtime Permissions Code Lab!

The code that the presenter will be walking through comes from the upcoming Version 6.9 of the award-seeking book, *The Busy Coder's Guide to Android Development*. This PDF file contains the tutorial chapter from that book that goes along with the code, showing the steps of how to convert a sample Android project to one that uses Android 6.0's runtime permissions model.

Like the Tutorial? Want More?

The Busy Coder's Guide to Android Development (https://commonsware.com/Android) is far more than just this one tutorial. Spanning nearly 3,400 pages, spread over 200+ chapters, this book is the first, largest, and most up-to-date book on Android development that you can get. The book is available as part of the Warescription program, where for $45, you get the book (in PDF, EPUB, or MOBI/Kindle formats, plus as an Android app), updates to the book for a year, access to office hours chats for getting your Android development questions answered, and more!

Like the Code Lab? Want More?

Mark Murphy – the balding guy who is leading the code lab – not only writes the aforementioned book, but also teaches Android app development. Introductory courses are available for your organization, and advanced one-day seminars are held in New York City:

- Updating Your App for Android 6.0 on September 29
- Security and Device Administration on September 30 and January 20
- Performance Analysis and Tuning on November 4
- Media and TV on November 5

Visit https://commonsware.com/training to learn more about CommonsWare's training options.

Tutorial: Runtime Permission Support

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:

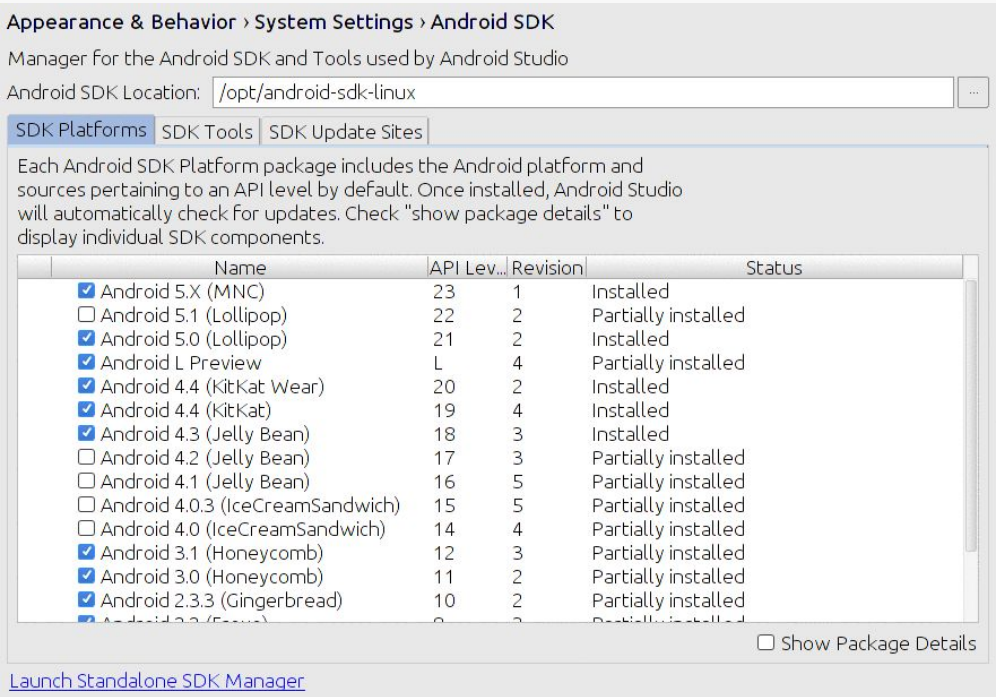


Figure 238: Android Studio 1.3 SDK Manager, Sans Android 6.0

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:

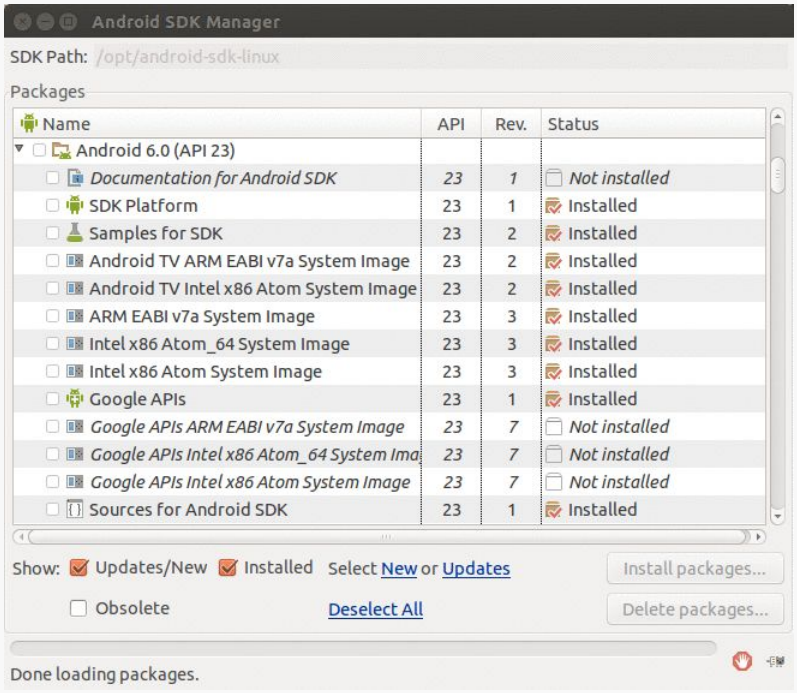


Figure 239: Classic SDK Manager, Showing Android 6.0

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:

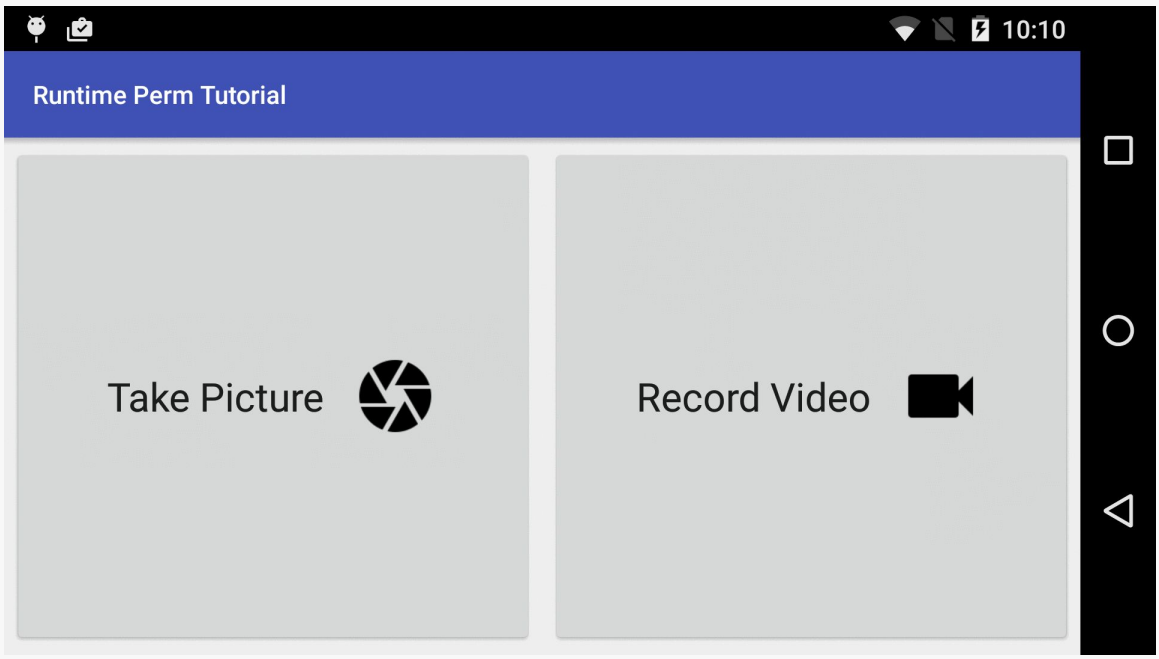


Figure 240: Runtime Permissions Tutorial App, As Initially Written and Launched

Tapping the "Take Picture" button will bring up a camera preview, with a floating action button (FAB) to take a picture:

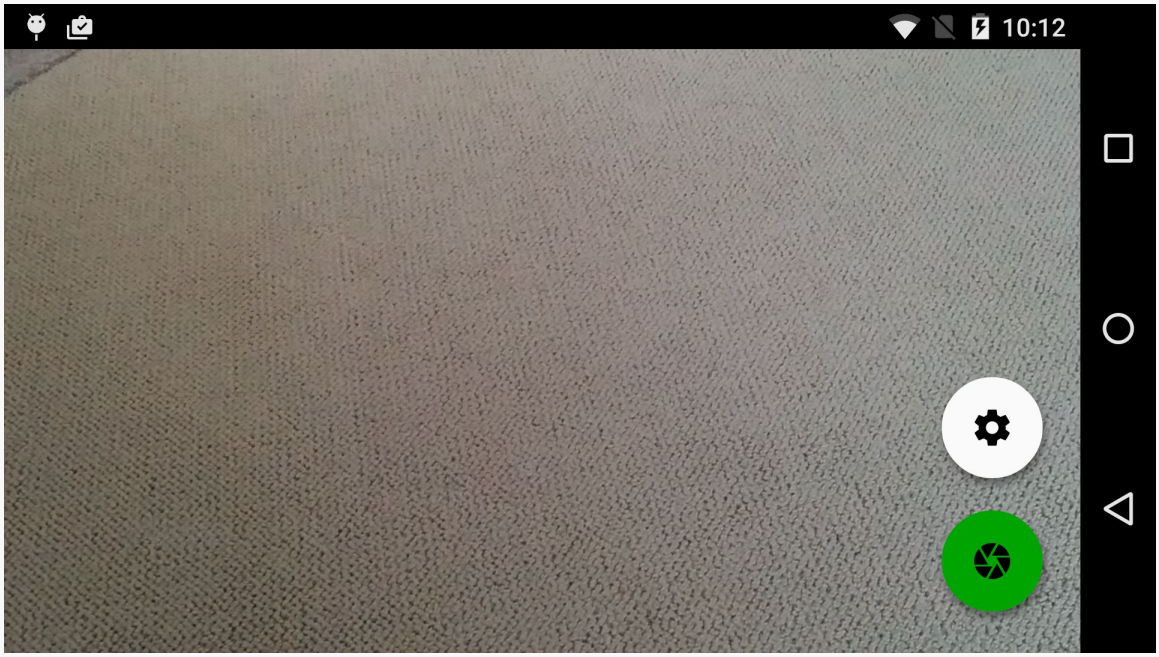


Figure 241: Runtime Permissions Tutorial App, Showing Camera Preview

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:

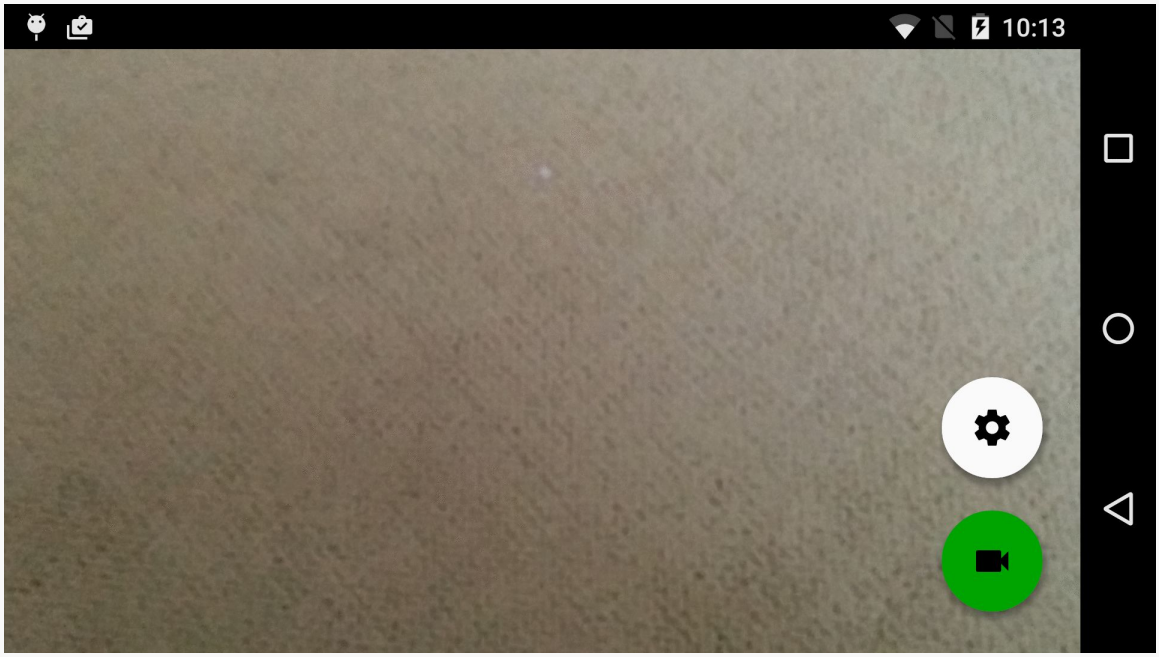


Figure 242: Runtime Permissions Tutorial App, Showing Video Preview

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

```
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'
}

android {
  compileSdkVersion 22
  buildToolsVersion "22.0.1"

  defaultConfig {
    minSdkVersion 15
```

```
    targetSdkVersion 15
  }
}
```

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:

```
<?xml version="1.0" encoding="utf-8"?>
<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">

  <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"
    android:layout_weight="1"
    android:drawableRight="@drawable/ic_videocam_black_48dp"
    android:onClick="recordVideo"
    android:text="Record Video"
    android:textAppearance="?android:attr/textAppearanceLarge"
    app:iconPadding="16dp"/>
</LinearLayout>
```

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

```
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;
  private View takePicture;
  private View recordVideo;

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

    takePicture=findViewById(R.id.take_picture);
    recordVideo=findViewById(R.id.record_video);

    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 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: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:

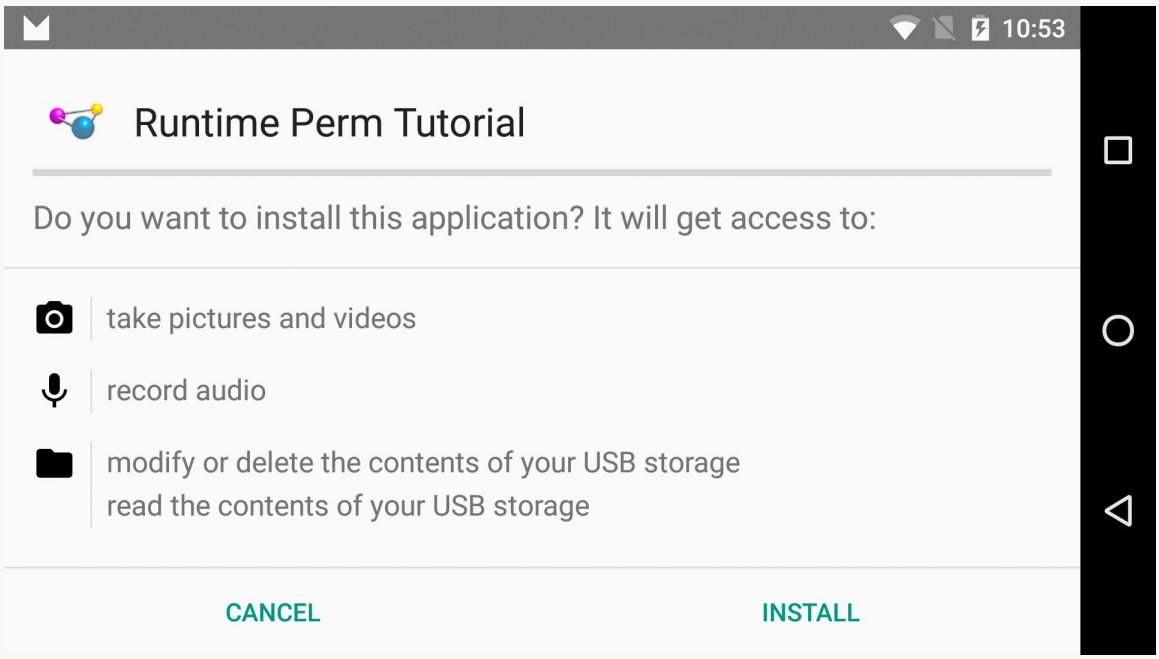


Figure 243: Installing the Tutorial App From the Web

However, the user can still go into Settings and elect to disable our access to those permissions:

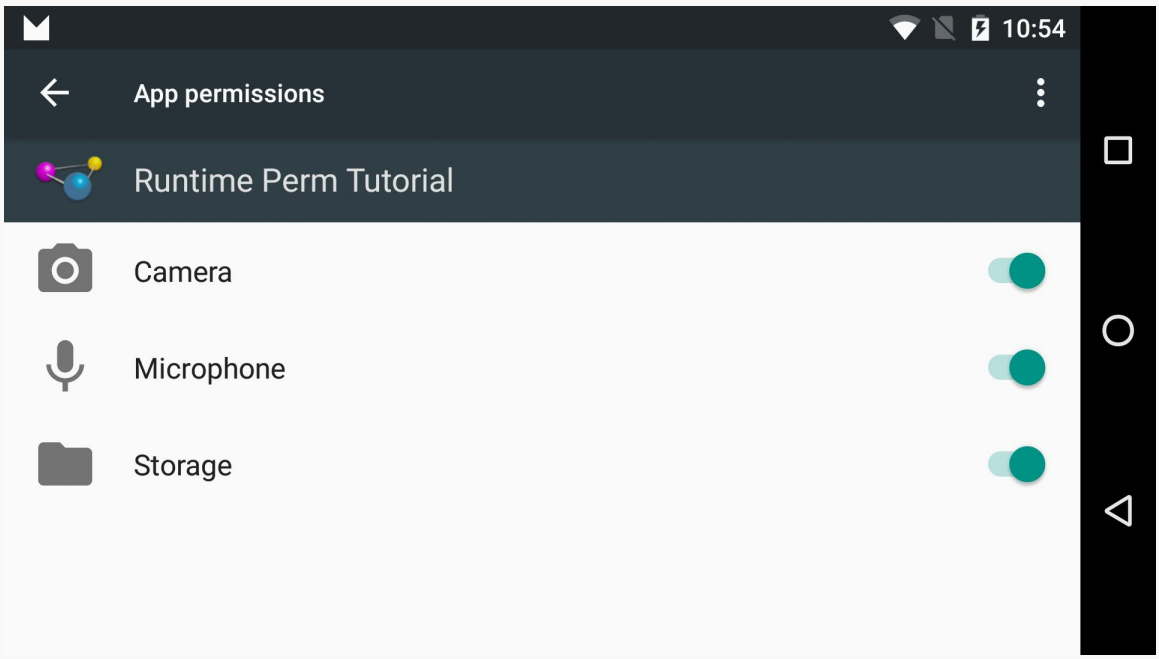


Figure 244: Settings, On Android 6.0, Showing Tutorial App Permissions

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:

```
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`:

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

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

Then, add the following data member to `MainActivity`:

```
  private SharedPreferences prefs;
```

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

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

Then, add the following method to `MainActivity`:

```
  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);
  }
```

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:

```
    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`:

```
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:

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

Also add the following `int` constant to `MainActivity`:

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

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

```
    if (isFirstRun()) {
      requestPermissions(PERMS_TAKE_PICTURE,
        RESULT_PERMS_INITIAL);
    }
```

Here, we are asking Android to collect our permissions, if this is the first run of our app.

However, we have a problem: this app's `minSdkVersion` is 15. We cannot call `requestPermissions()` on older devices, as that method does not exist. In fact, a lot of what we are doing in this tutorial will only be relevant on API Level 23+ devices.

There are two ways of addressing this problem:

1. Use `ActivityCompat.requestPermissions()`, where `ActivityCompat` is from the `support-v4` artifact, or
2. Use `RuntimePermissionUtils`, from the author's CWAC-Security library

As this tutorial will use other things from `RuntimePermissionUtils` later on, we will go with that approach.

Update your `dependencies` in your `app/` module's `build.gradle` file to include a reference to CWAC-Security:

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

```
    compile 'com.commonsware.cwac:security:0.6.+'
}
```

Next, add a data member for a `RuntimePermissionUtils` instance:

```
  private RuntimePermissionUtils utils;
```

Then, initialize that instance somewhere early in `onCreate()`:

```
    utils=new RuntimePermissionUtils(this);
```

Next, amend the `if` block in `onCreate()` to use the `utils` object to confirm that we are on a version of Android where runtime permissions are relevant:

```
    if (isFirstRun() && utils.useRuntimePermissions()) {
      requestPermissions(PERMS_TAKE_PICTURE,
        RESULT_PERMS_INITIAL);
    }
```

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

```
  @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:

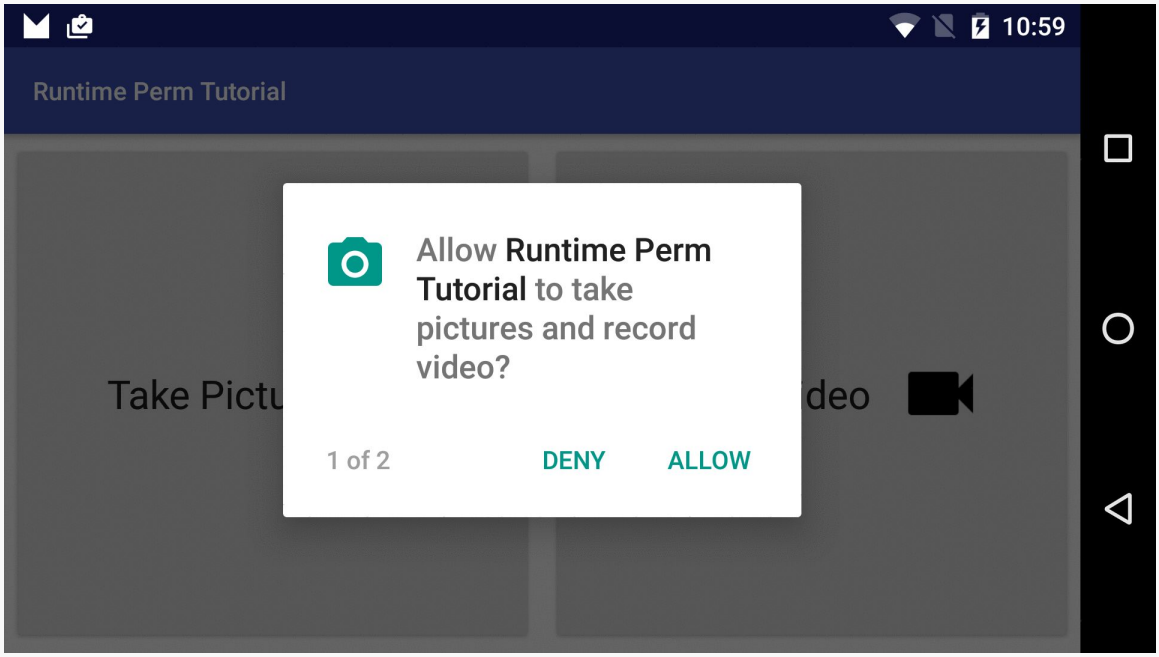


Figure 245: Tutorial App, Showing CAMERA Permission Request

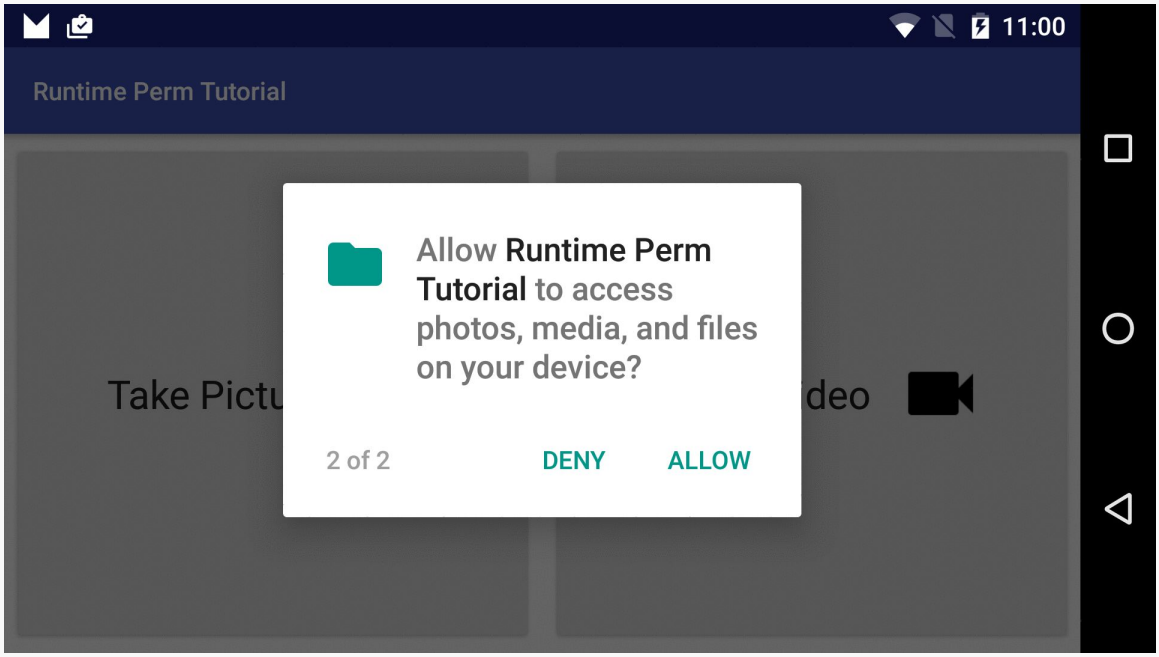


Figure 246: Tutorial App, Showing WRITE_EXTERNAL_STORAGE Permission Request

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 to `MainActivity`:

```
  private boolean canTakePicture() {
    return(utils.hasPermission(CAMERA) &&
      utils.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. It uses the `RuntimePermissionUtils` object to do this, where that method, under the covers, uses `checkSelfPermission()`, if we are on a compatible version of Android.

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

```
  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`:

```
  private boolean shouldShowTakePictureRationale() {
    return(utils.shouldShowRationale(this, CAMERA) ||
      utils.shouldShowRationale(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 `showShowRationale()` on `RuntimePermissionUtils`, where that method uses `shouldShowTakePictureRationale()`, along with `checkSelfPermission()` (as we do not need to show rationale if we already hold the permission) and `useRuntimePermissions()` (as none of this is relevant on older Android devices).

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

```
  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 `breadcrust`.

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

```
<?xml version="1.0" encoding="utf-8"?>
<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"
    android:onClick="recordVideo"
    android:text="Record Video"
    android:textAppearance="?android:attr/textAppearanceLarge"
    app:iconPadding="16dp"/>
</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 version="1.0" encoding="utf-8"?>
<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"
    android:layout_weight="1"
    android:drawableRight="@drawable/ic_videocam_black_48dp"
    android:onClick="recordVideo"
    android:text="Record Video"
    android:textAppearance="?android:attr/textAppearanceLarge"
    app:iconPadding="16dp"/>
</LinearLayout>
```

Next, add a data member for the `breadcrust` to `MainActivity`:

```
  private TextView breadcrust;
```

Then, in `onCreate()` of `MainActivity`, after the calls to `findViewById()` to look up the `takePicture` and `recordVideo` buttons, add a third call to `findViewById()` to look up the `breadcrust`:

```
    takePicture=findViewById(R.id.take_picture);
    recordVideo=findViewById(R.id.record_video);
    breadcrust=(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 `breadcrust` when we are going to ask the user (again) for permission to take pictures:

```
    <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 `breadcrust` (and so can see our rationale for requesting the permission), request the permissions again
- If we do not have permission, and the `breadcrust` is not visible, then we need to show the `breadcrust` with our rationale message

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

```
public void takePicture(View v) {
  if (canTakePicture()) {
    takePictureForRealz();
  }
  else if (breadcrust.getVisibility()==View.VISIBLE) {
    breadcrust.setVisibility(View.GONE);
    requestPermissions(utils.netPermissions(PERMS_TAKE_PICTURE),
        RESULT_PERMS_TAKE_PICTURE);
  }
  else if (shouldShowTakePictureRationale()) {
    breadcrust.setText(R.string.msg_take_picture);
    breadcrust.setVisibility(View.VISIBLE);
  }
  else {
    throw new IllegalStateException(getString(R.string.msg_state));
  }
}
```

If `breadcrust` is visible, we make it `GONE` again and call `requestPermissions`. If `breadcrust` is not visible, we make it `VISIBLE` and set its message to the string resource that we defined.

Along the way, we use a `netPermissions()` method on `RuntimePermissionUtils`. 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()`.

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

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

Your IDE will also yell about `R.string.msg_state` not being defined, so add that to `res/values/strings.xml`:

```
    <string name="msg_state">And you may ask yourself: well... how did I get
here?</string>
```

If we call `requestPermissions()` and the user grants the permissions, we should go ahead and take the picture. To do that, we need to add some more logic to the `onRequestPermissionsResult()` callback method in `MainActivity`, so alter yours to look like this:

```
  @Override
  public void onRequestPermissionsResult(int requestCode,
                                         String[] permissions,
                                         int[] grantResults) {
    if (requestCode==RESULT_PERMS_TAKE_PICTURE) {
      if (canTakePicture()) {
        takePictureForRealz();
      }
    }
  }
```

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.

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:

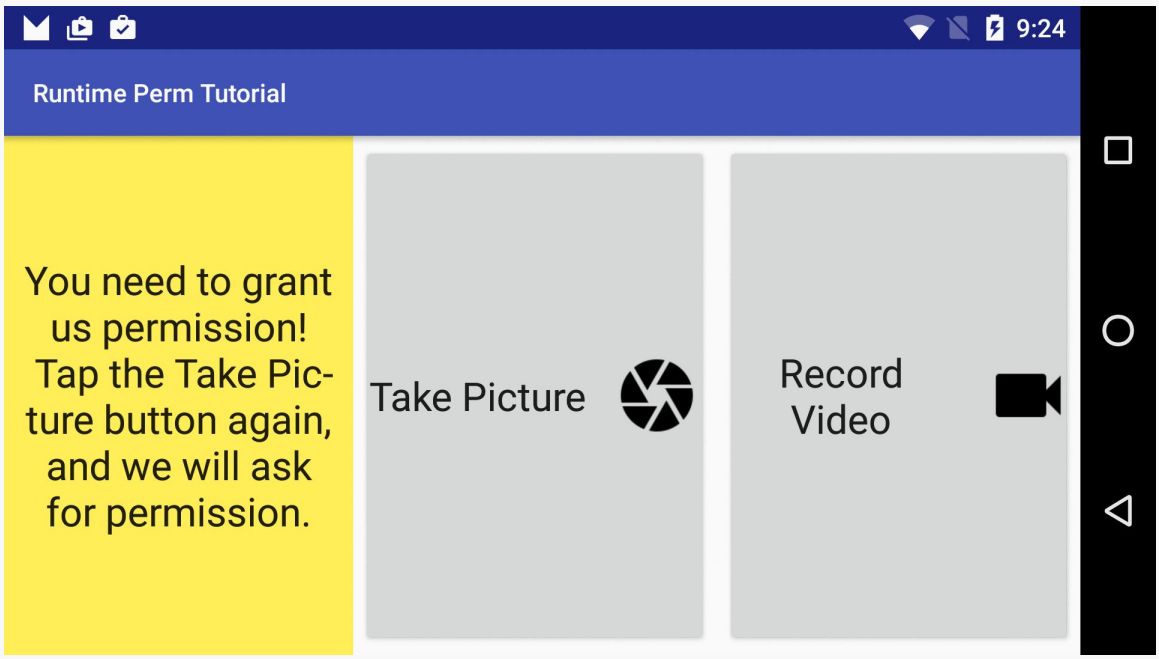


Figure 247: Tutorial App, Showing Breadcrust

If you tap the "Take Picture" button again, the `breadcrust` 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.

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`:

```
  private boolean canRecordVideo() {
    return(canTakePicture() && utils.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:

```
  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 `breadcrust`, 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`:

```
  private boolean shouldShowRecordVideoRationale() {
    return(shouldShowTakePictureRationale() ||
      utils.shouldShowRationale(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:

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

```
      breadcrust.setText(R.string.msg_record_video);
      breadcrust.setVisibility(View.VISIBLE);
    }
    else {
      throw new IllegalStateException(getString(R.string.msg_state));
    }
  }
```

This will require you to add another string resource to `res/values/strings.xml`:

```
    <string name="msg_record_video">You need to grant us permission! Tap the Record
Video button again, and we will ask for permission.</string>
```

Step #11: (Re-)Request Permissions

Of course, we still do not have any code to ask for the `RECORD_AUDIO` permission.

Importantly, `shouldShowRequestPermissionRationale()` will return `false` in two scenarios:

1. We asked the user for permission and the user declined, while checking the "don't ask again" checkbox
2. We never asked the user for the permission

Ideally, Android would help us distinguish between those two cases. Alas, Android does not. So, we need to track that ourselves. Fortunately, `RuntimePermissionUtils` contains two convenience methods for this:

- `haveEverRequestedPermission()`, which returns `true` or `false` depending upon whether we have ever requested the supplied permission, and
- `markPermissionAsRequested()`, to ensure that `haveEverRequestedPermission()` will return `true` in all future calls

This information is stored in a library-specific `SharedPreferences` file.

With the assistance of those two methods, we can complete the logic for `recordVideo()` in `MainActivity`:

```
  public void recordVideo(View v) {
    if (canRecordVideo()) {
      recordVideoForRealz();
    }
    else if (!utils.haveEverRequestedPermission(RECORD_AUDIO) ||
        breadcrust.getVisibility()==View.VISIBLE) {
      breadcrust.setVisibility(View.GONE);
      utils.markPermissionAsRequested(RECORD_AUDIO);
```

```
      requestPermissions(utils.netPermissions(PERMS_ALL),
          RESULT_PERMS_RECORD_VIDEO);
    }
    else if (shouldShowRecordVideoRationale()) {
      breadcrust.setText(R.string.msg_record_video);
      breadcrust.setVisibility(View.VISIBLE);
    }
    else {
      throw new IllegalStateException(getString(R.string.msg_state));
    }
  }
```

Here, we will request our permissions if either we have never asked before or the `breadcrust` is showing. Note that we are not distinguishing between the two uses of the `breadcrust` — if the user taps "Take Picture" and the `breadcrust` appears, and then the user taps "Record Video", we will ask for the permissions needed for recording a 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;
```

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

```
  @Override
  public void onRequestPermissionsResult(int requestCode,
                                         String[] permissions,
                                         int[] grantResults) {
    if (requestCode==RESULT_PERMS_TAKE_PICTURE) {
      if (canTakePicture()) {
        takePictureForRealz();
      }
    }
    else if (requestCode==RESULT_PERMS_RECORD_VIDEO) {
      if (canRecordVideo()) {
        recordVideoForRealz();
      }
    }
  }
```

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. Then, uninstall the app.

Step #12: Update the Button Status

If through "Take Picture" or "Record Video" taps, when you are prompted to grant permissions, you check the "don't ask again" checkbox... you crash with an `IllegalStateException`, which is not exactly ideal. Basically, we fall through all the cases in `takePicture()` or `recordVideo()` and hit the final `else` block. We really should fix that.

The best way to fix it would be to disable the buttons once there is no hope of getting the permissions, short of the user visiting the Settings app and toggling on those permissions for us.

`RuntimePermissionUtils` comes to the rescue again, with a `wasPermissionRejected()` method. `wasPermissionRejected()`, for a given permission, will return `true` if we do not hold the permission and we should not show the rationale for the permission. `true` would mean that we have no means of getting the permission. The exception is if we never asked for the permission in the first place, which we will have to deal with for `RECORD_AUDIO` ourselves.

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

```
  private boolean couldPossiblyTakePicture() {
    return(!utils.wasPermissionRejected(this, CAMERA) &&
        !utils.wasPermissionRejected(this,
          WRITE_EXTERNAL_STORAGE));
  }
```

`couldPossiblyTakePicture()` will return `true` if both the `CAMERA` or `WRITE_EXTERNAL_STORAGE` permissions are still in play. It will return `false` if either one of them can no longer be obtained.

Then, add the following method to `MainActivity`:

```
  private boolean wasAudioRejected() {
    return(!utils.hasPermission(RECORD_AUDIO) &&
            !utils.shouldShowRationale(this, RECORD_AUDIO) &&
```

```
            utils.haveEverRequestedPermission(RECORD_AUDIO));
  }
```

`wasAudioRejected()` is a permission-specific rendition of `wasPermissionRejected()` with one big difference: we also see if we have ever asked for this permission, via the `haveRequestedAudioPermission()` method we added earlier.

Now we can add a `couldPossiblyRecordVideo()` method to `MainActivity` to use that:

```
  private boolean couldPossiblyRecordVideo() {
    return(couldPossiblyTakePicture() && !wasAudioRejected());
  }
```

Next, we can use all of this to disable our buttons, via an `updateButtons()` method that you can add to `MainActivity`:

```
  private void updateButtons() {
    takePicture.setEnabled(couldPossiblyTakePicture());
    recordVideo.setEnabled(couldPossiblyRecordVideo());
  }
```

Finally, we need to call that `updateButtons()` method. One obvious place to call it is in `onRequestPermissionsResult()`, as we know that there is a decent chance that our mix of available permissions will have changed. So, add a call to `updateButtons()` to `onRequestPermissionsResult()` of `MainActivity`:

```
  @Override
  public void onRequestPermissionsResult(int requestCode,
                                         String[] permissions,
                                         int[] grantResults) {
    updateButtons();

    if (requestCode==RESULT_PERMS_TAKE_PICTURE) {
      if (canTakePicture()) {
        takePictureForRealz();
      }
    }
    else if (requestCode==RESULT_PERMS_RECORD_VIDEO) {
      if (canRecordVideo()) {
        recordVideoForRealz();
      }
    }
  }
```

We also need to handle our initial state, when we launch the app, particularly for when we launch the app the second or later times (and so will not be asking for permissions up front). `onResume()` is a reasonable spot for that, so add an `onResume()` method to `MainActivity`:

```
@Override
protected void onResume() {
  super.onResume();

  updateButtons();  // Settings does not terminate process
                    // if permission granted, only if revoked
}
```

The code comment is a reminder that while the Settings app will terminate our process if the user revokes one of our permissions, Settings will not terminate our process if the user grants one of our permissions that was previously declined or revoked. This is another reason to check in `onResume()` (versus `onCreate()`): the user might leave our app, go to Settings, grant us our permissions, and return to our app.

If you install the app, reject the permissions at the outset, tap either button, and reject the permissions a second time, checking the "don't ask again" checkbox, the buttons should disable. Then, uninstall the app.

Step #13: 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`:

```
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`:

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

  if (breadcrust.getVisibility()==View.VISIBLE) {
    outState.putCharSequence(STATE_BREADCRUST,
      breadcrust.getText());
  }
```

```
  }

  @Override
  protected void onRestoreInstanceState(Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);

    CharSequence cs=savedInstanceState.getCharSequence(STATE_BREADCRUST);

    if (cs!=null) {
      breadcrust.setVisibility(View.VISIBLE);
      breadcrust.setText(cs);
    }
  }
```

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

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 `breadcrust` to appear, rotating the device or otherwise triggering a configuration change will not lose the `breadcrust`, even though our activity will be destroyed and recreated along the way.

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

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

import android.app.Activity;
import android.content.Intent;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.os.Environment;
import android.preference.PreferenceManager;
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 com.commonsware.cwac.security.RuntimePermissionUtils;
import java.io.File;
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_RECORDED=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 View takePicture;
  private View recordVideo;
  private TextView breadcrust;
  private SharedPreferences prefs;
  private RuntimePermissionUtils utils;

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

    utils=new RuntimePermissionUtils(this);
    prefs=PreferenceManager.getDefaultSharedPreferences(this);

    takePicture=findViewById(R.id.take_picture);
    recordVideo=findViewById(R.id.record_video);
    breadcrust=(TextView)findViewById(R.id.breadcrust);

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

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

    if (isFirstRun() && utils.useRuntimePermissions()) {
      requestPermissions(PERMS_TAKE_PICTURE,
        RESULT_PERMS_INITIAL);
    }
  }

  @Override
  protected void onResume() {
    super.onResume();

    updateButtons();  // Settings does not terminate process
                      // if permission granted, only if revoked
  }
```

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

  if (breadcrust.getVisibility()==View.VISIBLE) {
    outState.putCharSequence(STATE_BREADCRUST,
      breadcrust.getText());
  }
}

@Override
protected void onRestoreInstanceState(Bundle savedInstanceState) {
  super.onRestoreInstanceState(savedInstanceState);

  CharSequence cs=savedInstanceState.getCharSequence(STATE_BREADCRUST);

  if (cs!=null) {
    breadcrust.setVisibility(View.VISIBLE);
    breadcrust.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_RECORDED) {
      t=Toast.makeText(this, R.string.msg_vid_recorded,
          Toast.LENGTH_LONG);
    }

    t.show();
  }
}

@Override
public void onRequestPermissionsResult(int requestCode,
                                       String[] permissions,
                                       int[] grantResults) {
  updateButtons();

  if (requestCode==RESULT_PERMS_TAKE_PICTURE) {
    if (canTakePicture()) {
      takePictureForRealz();
    }
  }
  else if (requestCode==RESULT_PERMS_RECORD_VIDEO) {
    if (canRecordVideo()) {
      recordVideoForRealz();
    }
  }
```

```
}

public void takePicture(View v) {
  if (canTakePicture()) {
    takePictureForRealz();
  }
  else if (breadcrust.getVisibility()==View.VISIBLE) {
    breadcrust.setVisibility(View.GONE);
    requestPermissions(utils.netPermissions(PERMS_TAKE_PICTURE),
        RESULT_PERMS_TAKE_PICTURE);
  }
  else if (shouldShowTakePictureRationale()) {
    breadcrust.setText(R.string.msg_take_picture);
    breadcrust.setVisibility(View.VISIBLE);
  }
  else {
    throw new IllegalStateException(getString(R.string.msg_state));
  }
}

public void recordVideo(View v) {
  if (canRecordVideo()) {
    recordVideoForRealz();
  }
  else if (!utils.haveEverRequestedPermission(RECORD_AUDIO) ||
      breadcrust.getVisibility()==View.VISIBLE) {
    breadcrust.setVisibility(View.GONE);
    utils.markPermissionAsRequested(RECORD_AUDIO);
    requestPermissions(utils.netPermissions(PERMS_ALL),
        RESULT_PERMS_RECORD_VIDEO);
  }
  else if (shouldShowRecordVideoRationale()) {
    breadcrust.setText(R.string.msg_record_video);
    breadcrust.setVisibility(View.VISIBLE);
  }
  else {
    throw new IllegalStateException(getString(R.string.msg_state));
  }
}

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 void updateButtons() {
  takePicture.setEnabled(couldPossiblyTakePicture());
  recordVideo.setEnabled(couldPossiblyRecordVideo());
}

private boolean wasAudioRejected() {
  return(!utils.hasPermission(RECORD_AUDIO) &&
         !utils.shouldShowRationale(this, RECORD_AUDIO) &&
```

```
            utils.haveEverRequestedPermission(RECORD_AUDIO));
  }

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

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

  private boolean shouldShowTakePictureRationale() {
    return(utils.shouldShowRationale(this, CAMERA) ||
      utils.shouldShowRationale(this, WRITE_EXTERNAL_STORAGE));
  }

  private boolean shouldShowRecordVideoRationale() {
    return(shouldShowTakePictureRationale() ||
      utils.shouldShowRationale(this, RECORD_AUDIO));
  }

  private boolean couldPossiblyTakePicture() {
    return(!utils.wasPermissionRejected(this, CAMERA) &&
        !utils.wasPermissionRejected(this,
          WRITE_EXTERNAL_STORAGE));
  }

  private boolean couldPossiblyRecordVideo() {
    return(couldPossiblyTakePicture() && !wasAudioRejected());
  }

  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);
  }
}
```

And your tutorial is now complete.