Taking Photos Simply

This lesson explains how to capture photos using an existing camera application.

Suppose you are implementing a crowd-sourced weather service that makes a global weather map by blending together pictures of the sky taken by devices running your client app. Integrating photos is only a small part of your application. You want to take photos with minimal fuss, not reinvent the camera. Happily, most Android-powered devices already have at least one camera application installed. In this lesson, you learn how to make it take a picture for you.

Request Camera Permission

If an essential function of your application is taking pictures, then restrict its visibility on Google Play to devices that have a camera. To advertise that your application depends on having a camera, put a `<uses-feature>` tag in your manifest file:

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

If your application uses, but does not require a camera in order to function, instead set `android:required` to `false`. In doing so, Google Play will allow devices without a camera to download your application. It's then your responsibility to check for the availability of the camera at runtime by calling `hasSystemFeature(PackageManager.FEATURE_CAMERA)`. If a camera is not available, you should then disable your camera features.

Take a Photo with the Camera App

The Android way of delegating actions to other applications is to invoke an `Intent` that describes what you want done. This process involves three pieces: The `Intent` itself, a call to start the external `Activity`, and some code to handle the image data when focus returns to your activity.

Here's a function that invokes an intent to capture a photo.

```
static final int REQUEST_IMAGE_CAPTURE = 1;

private void dispatchTakePictureIntent() {
    Intent takePictureIntent = new
Intent(MediaStore.ACTION_IMAGE_CAPTURE);
```

```
    if (takePictureIntent.resolveActivity(getPackageManager()) != null) {
        startActivityForResult(takePictureIntent, REQUEST_IMAGE_CAPTURE);
    }
}
```

Notice that the `startActivityForResult()` method is protected by a condition that calls `resolveActivity()`, which returns the first activity component that can handle the intent. Performing this check is important because if you call `startActivityForResult()` using an intent that no app can handle, your app will crash. So as long as the result is not null, it's safe to use the intent.

Get the Thumbnail

If the simple feat of taking a photo is not the culmination of your app's ambition, then you probably want to get the image back from the camera application and do something with it.

The Android Camera application encodes the photo in the return `Intent` delivered to `onActivityResult()` as a small `Bitmap` in the extras, under the key `"data"`. The following code retrieves this image and displays it in an `ImageView`.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent
data) {
    if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode == RESULT_OK) {
        Bundle extras = data.getExtras();
        Bitmap imageBitmap = (Bitmap) extras.get("data");
        mImageView.setImageBitmap(imageBitmap);
    }
}
```

Note: This thumbnail image from `"data"` might be good for an icon, but not a lot more. Dealing with a full-sized image takes a bit more work.

Save the Full-size Photo

The Android Camera application saves a full-size photo if you give it a file to save into. You must provide a fully qualified file name where the camera app should save the photo.

Generally, any photos that the user captures with the device camera should be saved on the device in the public external storage so they are accessible by all apps. The proper directory for shared photos is provided by `getExternalStoragePublicDirectory()`, with the `DIRECTORY_PICTURES` argument. Because the directory provided by this method is shared among all apps, reading and writing to it requires the `READ_EXTERNAL_STORAGE` and `WRITE_EXTERNAL_STORAGE` permissions,

respectively. The write permission implicitly allows reading, so if you need to write to the external storage then you need to request only one permission:

```
<manifest ...>
    <uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    ...
</manifest>
```

However, if you'd like the photos to remain private to your app only, you can instead use the directory provided by `getExternalFilesDir()`. On Android 4.3 and lower, writing to this directory also requires the `WRITE_EXTERNAL_STORAGE` permission. Beginning with Android 4.4, the permission is no longer required because the directory is not accessible by other apps, so you can declare the permission should be requested only on the lower versions of Android by adding the `maxSdkVersion` attribute:

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

Note: Files you save in the directories provided by `getExternalFilesDir()` are deleted when the user uninstalls your app.

Once you decide the directory for the file, you need to create a collision-resistant file name. You may wish also to save the path in a member variable for later use. Here's an example solution in a method that returns a unique file name for a new photo using a date-time stamp:

```
String mCurrentPhotoPath;

private File createImageFile() throws IOException {
    // Create an image file name
    String timeStamp = new SimpleDateFormat("yyyyMMdd_HHmmss").format(new
Date());
    String imageFileName = "JPEG_" + timeStamp + "_";
    File storageDir = Environment.getExternalStoragePublicDirectory(
            Environment.DIRECTORY_PICTURES);
    File image = File.createTempFile(
        imageFileName,  /* prefix */
        ".jpg",         /* suffix */
        storageDir      /* directory */
```

```
    );

    // Save a file: path for use with ACTION_VIEW intents
    mCurrentPhotoPath = "file:" + image.getAbsolutePath();
    return image;
}
```

With this method available to create a file for the photo, you can now create and invoke the `Intent` like this:

```
static final int REQUEST_TAKE_PHOTO = 1;

private void dispatchTakePictureIntent() {
    Intent takePictureIntent = new
Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    // Ensure that there's a camera activity to handle the intent
    if (takePictureIntent.resolveActivity(getPackageManager()) != null) {
        // Create the File where the photo should go
        File photoFile = null;
        try {
            photoFile = createImageFile();
        } catch (IOException ex) {
            // Error occurred while creating the File
            ...
        }
        // Continue only if the File was successfully created
        if (photoFile != null) {
            takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT,
                    Uri.fromFile(photoFile));
            startActivityForResult(takePictureIntent, REQUEST_TAKE_PHOTO);
        }
    }
}
```

Add the Photo to a Gallery

When you create a photo through an intent, you should know where your image is located, because you said where to save it in the first place. For everyone else, perhaps the easiest way to make your photo accessible is to make it accessible from the system's Media Provider.

Note: If you saved your photo to the directory provided by `getExternalFilesDir()`, the media scanner cannot access the files because they are private to your app.

The following example method demonstrates how to invoke the system's media scanner to add your photo to the Media Provider's database, making it available in the Android Gallery application and to other apps.

```
private void galleryAddPic() {
    Intent mediaScanIntent = new
Intent(Intent.ACTION_MEDIA_SCANNER_SCAN_FILE);
    File f = new File(mCurrentPhotoPath);
    Uri contentUri = Uri.fromFile(f);
    mediaScanIntent.setData(contentUri);
    this.sendBroadcast(mediaScanIntent);
}
```

Decode a Scaled Image

Managing multiple full-sized images can be tricky with limited memory. If you find your application running out of memory after displaying just a few images, you can dramatically reduce the amount of dynamic heap used by expanding the JPEG into a memory array that's already scaled to match the size of the destination view. The following example method demonstrates this technique.

```
private void setPic() {
    // Get the dimensions of the View
    int targetW = mImageView.getWidth();
    int targetH = mImageView.getHeight();

    // Get the dimensions of the bitmap
    BitmapFactory.Options bmOptions = new BitmapFactory.Options();
    bmOptions.inJustDecodeBounds = true;
    BitmapFactory.decodeFile(mCurrentPhotoPath, bmOptions);
    int photoW = bmOptions.outWidth;
    int photoH = bmOptions.outHeight;

    // Determine how much to scale down the image
    int scaleFactor = Math.min(photoW/targetW, photoH/targetH);

    // Decode the image file into a Bitmap sized to fill the View
    bmOptions.inJustDecodeBounds = false;
    bmOptions.inSampleSize = scaleFactor;
    bmOptions.inPurgeable = true;

    Bitmap bitmap = BitmapFactory.decodeFile(mCurrentPhotoPath,
bmOptions);
    mImageView.setImageBitmap(bitmap);
```

```
}
```