IN721 2019 – QRCode Reader

## To build a QR Code reader you need two elements:

* 1. A CameraSource. A class that captures input frames from the camera.
  2. A Detector. A class that implements logic to process camera frames.

## You then:

1. Create and configure the Detector.
2. Create and configure the CameraSource, passing it the prepared Detector.

Create the detector:

You can write custom detectors, but there are some very powerful ones already available in the Android libraries. We are going to use the detector class which knows how to parse bar codes and QR codes. To make an instance of this class, we will use the associated Builder, as we have seen in Android before.

BarcodeDetector **barcodeDetector**;

BarcodeDetector.Builder detectorBuilder = **new** BarcodeDetector.Builder(**this**);  
detectorBuilder.setBarcodeFormats(Barcode.***QR\_CODE***);

**barcodeDetector** = detectorBuilder.build();

Configure the detector.

Once we have our BarcodeDetector instance, we need to tell it what to do when it detects something. This is logically equivalent to the way that we tell a button what to do when it is clicked. With buttons, we use the button's setOnClickListener method to pass in an object that holds its click handler code. With BarcodeDetectors, we will use the setProcessor method to pass in an object that holds its "something has been detected" handler code.

As with other event handlers, this object must implement a particular interface, called Detector.Processor<Barcode>. This interface has two methods: release and receiveDetections. We won't use release for now. The receiveDetections event method is passed in a Detector.Detections object, which is basically an array of detected things. If we configure our BarcodeDetector as shown above, the "detected thing" will be a QR code.

Here is an example of the detection event handler class:

**public class** BarcodeDetectorProcessor **implements** Detector.Processor<Barcode>  
{  
 @Override  
 **public void** release() {}  
  
 @Override  
 **public void** receiveDetections(Detector.Detections<Barcode> detections)  
 {  
 *// Detector.getDetectedItems returns a SparseArray of all the things it has been sent from the camera  
 // SparseArray is just a different underlying data structure for arrays, which is more efficient for fetching* SparseArray<Barcode> barcodes = detections.getDetectedItems();  
  
 *// Check that something was detected, just in case....* **if** (barcodes.size() != 0)  
 {  
 *// This is how to get the QR Code text out of the "detected thing" class instance* String qrCodeMessage = barcodes.valueAt(0).**displayValue**;  
  
 *// Do whatever you want to do with that string here.....*

*// For this practical, return it to the Activity that launched you, via intent*

*// using putExtra and finish(), as you have done before.* }  
 } *// end receiveDetections*} *// end BarCodeDetectorProcessor*

Having defined our handler class "BarCodeDetectorProcessr" as above, we now bind it to the BarcodeDetector with its setProcessor command. The entire code for creating the detector is thus:

BarcodeDetector **barcodeDetector**;

BarcodeDetector.Builder detectorBuilder = **new** BarcodeDetector.Builder(**this**);  
detectorBuilder.setBarcodeFormats(Barcode.***QR\_CODE***);

**barcodeDetector** = detectorBuilder.build();

barcodeDetector.setProcessor(**new** BarcodeDetectorProcessor());

## Create the CameraSource

The CameraSource object also uses a Builder. Here we will illustrate the more compact builder syntax that you are likely to see in the wild:

CameraSource **cameraSource**;

**cameraSource** = **new** CameraSource  
 .Builder(**this**, **barcodeDetector**)  
 .setRequestedPreviewSize(640, 640)  
 .build();

Note that the BarcodeDetector instance (pre-configured with its Processor handler code) must be passed in to the Builder. The frames captured by the CameraSource will be passed to the BarcodeDetector for processing.

At this point, you can start QR Code processing with:

**cameraSource**.start();

The CameraSource object will start grabbing frames from the camera and passing them along for processing. When a QR Code is detected, the receiveDetections event is raised, and you will be able to pull the code out into a string variable (as shown above). However, the user won't actually be able to see anything of this happening. If you want the user to be able to see the camera frames as they are captured, you need to add a special widget called "SurfaceView" to your Activity layout, and connect it to the CameraSource.

## SurfaceView

SurfaceView is a screen widget, just like TextView or ImageView, except that it runs on its own thread. SurfaceView is very powerful, and it is worth spending some time exploring all the things it can be used for. In this practical, we use it to preview frames from CameraSource.

SurfaceView is a complex control with many components. The component which provides the actual display surface is its SurfaceHolder. The cameraSource.start() method we saw above can accept a SurfaceHolder as an input argument, and it will display its frames there.

Unfortunately, because SurfaceView runs on its own thread, we can't just pass its SurfaceHolder directly into our CameraSource in the main activity code. We have to make that connection in a callback that is executed when the SurfaceView thread gets fully activated. The steps are what you are probably coming to expect by this point:

* define a callback class instance
* get the SurfaceHolder
* bind it to an instance of the callback class

The callback class implements the interface SurfaceHolder.Callback. This interface has three methods: surfaceCreated, surfaceChanged and surfaceDestroyed, which are self-explanatory. We want to start the cameraSource when the surfaceHolder is created and stop it if the surfaceHolder is destroyed. So the code might look like this (assuming the variable cameraSource is our created CameraSource object, and is in scope).

**public class** SurfaceHolderSetUp **implements** SurfaceHolder.Callback  
{  
 @Override  
 *// The created surfaceHolder is passed in...* **public void** surfaceCreated(SurfaceHolder surfaceHolder)  
 {  
 **try** {  
 **cameraSource**.start(surfaceHolder);  
 }  
 **catch** (IOException ie)  
 {  
 Log.*e*(**"CAMERA SOURCE"**, ie.getMessage());  
 }  
 }  
  
 @Override  
 **public void** surfaceChanged(SurfaceHolder surfaceHolder, **int** i, **int** i1, **int** i2) {}  
  
 @Override  
 **public void** surfaceDestroyed(SurfaceHolder surfaceHolder)  
 {  
 **cameraSource**.stop();  
 }  
} *// end SurfaceCallBack*

Once we have written this class, we can drop a SurfaceView onto our layout (mine has id camera\_view), grab its SurfaceHolder and wire it up to the callback like this:

SurfaceView svCameraSurfaceView = (SurfaceView)findViewById(R.id.***camera\_view***);  
SurfaceHolder cameraSurfaceHolder = svCameraSurfaceView.getHolder();  
cameraSurfaceHolder.addCallback(**new** SurfaceHolderSetUp());

Now the user will be able to see the camera frames in the SurfaceView control.

## Permissions and Dependencies

1. Add the camera permission to the manifest.
2. Add Google Play Services to the dependencies area in build.gradle (Module.app). The provided code will work with this version (newer versions may require explicit checkPermission logic):

implementation **'com.google.android.gms:play-services:7.8.0'**

## Putting it Together:

Make two activities. The main activity needs a Button and a TextView. On button click, launch the second activity via StartActivityForResult. The second activity will contain all the QR Code logic, so it needs a SurfaceView widget in its layout. Use the above code to get the string from a detected QR Code, and return it to the main activity in the usual way. In the main activity, set up OnActivityResult to display the captured code in its TextView.