|  |
| --- |
| Android Tutorial – Part 7 |

|  |
| --- |
| 6-29-2018 |



Table of Contents

[Introduction 2](#_Toc518628332)

[Android Camera API 3](#_Toc518628333)

[Things to consider before using Camera in the app 3](#_Toc518628334)

[Camera Requirement in app 3](#_Toc518628335)

[Quick Picture or Customized Camera 3](#_Toc518628336)

[Storage 3](#_Toc518628337)

[ConstraintLayout in Android 5](#_Toc518628338)

[Constraints overview 5](#_Toc518628339)

[The view inspector 8](#_Toc518628340)

[Android's Bluetooth API 22](#_Toc518628341)

[How Bluetooth works; 22](#_Toc518628342)

[Components in Android Bluetooth API 23](#_Toc518628343)

[BluetoothAdapter class 23](#_Toc518628344)

[Bluetooth permissions 24](#_Toc518628345)

[Enable Bluetooth on a device 27](#_Toc518628346)

[Enable visibility of device over Bluetooth 33](#_Toc518628347)

[List Bluetooth devices 35](#_Toc518628348)

[Difference between a paired and a connected Bluetooth device 35](#_Toc518628349)

[Pair a Bluetooth Device 45](#_Toc518628350)

[Send Data to a Bluetooth Device 47](#_Toc518628351)

[Scan for Bluetooth devices 50](#_Toc518628352)

[References 52](#_Toc518628353)

# Introduction

This is the part seven of the android tutorial series. It is a continuation from last week. In order to follow this successfully, it is required to have,

* A basic understanding given about android in last session.
* The environment set up.
* The project created during last tutorial, opened in Android Studio.
* AVD or an Actual device ready for app deployment.

To catch up, in the last session (Android Tutorial Part 6),

* Using Emulator to Test Sensors
* Android Motion Sensors
  + Rotational Vector Sensor
  + Gyroscope
* Android Position Sensor
  + Orientation Sensor
* Creating a Custom View in Android
  + Using paint object
  + Using canvas
* The proximity sensor
* Android environment sensors
  + Ambient Temperature
  + Light
  + Pressure
  + Relative Humidity
* Temperature

Source code for the previous tutorial:-<https://github.com/nadee158/android_tutorial_part_6.git>

With that knowledge in hand, in this session below areas will be covered,

* Android Camera API
* ConstraintLayout in Android
* Android Bluetooth API
  + Components in Android Bluetooth API
  + Bluetooth Permissions
  + Enable/Disable Bluetooth on a device
  + Enable visibility of device over Bluetooth
  + List Bluetooth devices
  + Pair a Bluetooth device
  + Send Data to a Bluetooth devices
  + Scan for Bluetooth device

# Android Camera API

Most Android devices have at least one camera. Some devices have a front and a back facing camera.

The Android framework includes support for various cameras and camera features available on devices, allowing developers to capture pictures and videos in the applications.

The Android framework supports capturing images and video through the **android.hardware.camera2 API** or **camera Intent**

## Things to consider before using Camera in the app

Before enabling the application to use cameras on Android devices, we should consider about how the app intends to use this hardware feature.

### Camera Requirement in app

* Is the app totally dependent on Camera – app will not work without it?
  + If yes, the camera requirement should be declared in the manifest.

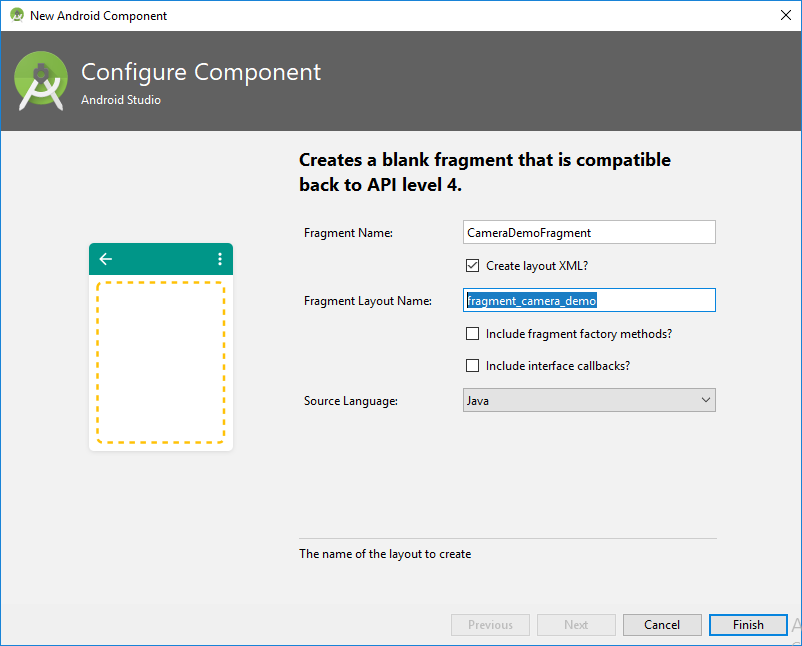
### Quick Picture or Customized Camera

* How will the application use the camera?
  + Is it for just snapping a quick picture or video clip?
    - If yes, consider Using Existing Camera Apps
  + Does the app provide a new way to use cameras?
    - If yes, build a camera using the Camera API

### Storage

The visibility, availability and security of the generated images or videos through the app

Lets now see how we could use an existing camera app to get an image to our app

1. Create a new “**Fragment**” to display the data retrieved from the Rotational Vector Sensor
   1. Right click on “**java/lk.uok.mit.fragment**”, select “**New**”🡪”**Fragment**”🡪”**Fragment (Blank)**”  
      
   2. Fill the details as shown below;  
      
      1. Fragment Name:- **CameraDemoFragment**
      2. Check create layout XML option
      3. Fragment Layout Name:- **fragment\_camera\_demo**
      4. **Uncheck both “Include fragment factory methods” and “include interface callback”**
      5. Source Language:-Java
2. Add a **menu item** to the **drawer** to access the new “**CameraDemoFragment**”
   1. Open the “**drawer\_view.xml**” inside “**res/menu**” folder and add a new menu item with id “**nav\_camera\_demo**” like below;  
      **<item  
       android:id="@+id/nav\_camera\_demo"  
       android:title="Camera Demo" />**
3. Add an entry to the switch case statement inside “**NavigationItemSelectedListener**” inside “**MainActivity**”
   1. Open “**MainActivity**” inside “**java**” folder
   2. Modify the “**setNavigationItemSelectedListene**r(” method of “**NavigationView**” inside “**onCreate**” method like shown below;  
        
      **case R.id.nav\_camera\_demo:  
       *//Open the CameraDemoFragment  
       // Begin the transaction* ft = getSupportFragmentManager().beginTransaction();  
       *// Replace the contents of the container with the new fragment* ft.replace(R.id.*fragment\_content*, new CameraDemoFragment());  
       *// Complete the changes added above* ft.commit();  
       break;**

# ConstraintLayout in Android

ConstraintLayout allows developers to create large and complex layouts with a **flat view hierarchy** (no nested view groups).

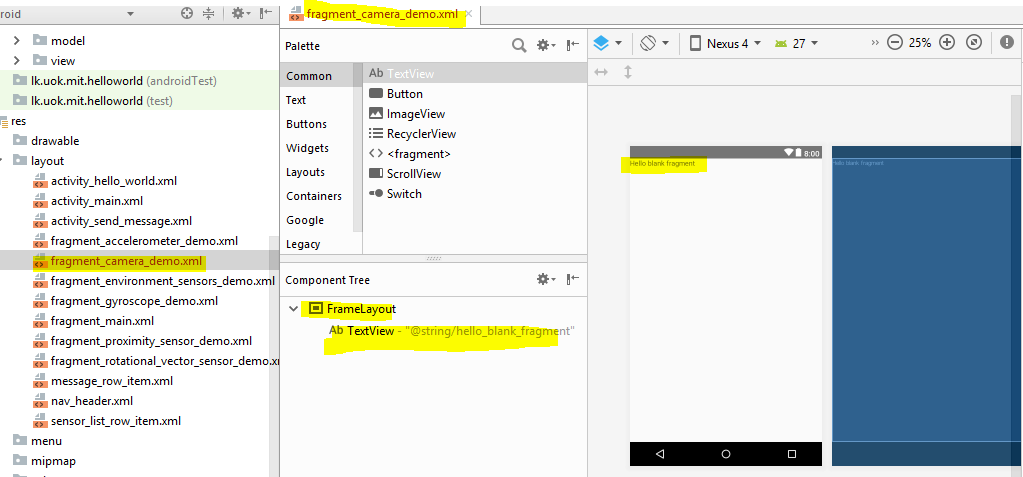
* It's **similar to RelativeLayout** in that all views are laid out according to relationships between sibling views and the parent layout,
* But it's **more flexible than RelativeLayout** and **easier to use with Android Studio's Layout Editor**.
  + All the power of ConstraintLayout is available directly from the Layout Editor's visual tools
  + The layout API and the Layout Editor were specially built for each other.
  + It is possible to build the layout with ConstraintLayout entirely by drag-and-dropping instead of editing the XML.

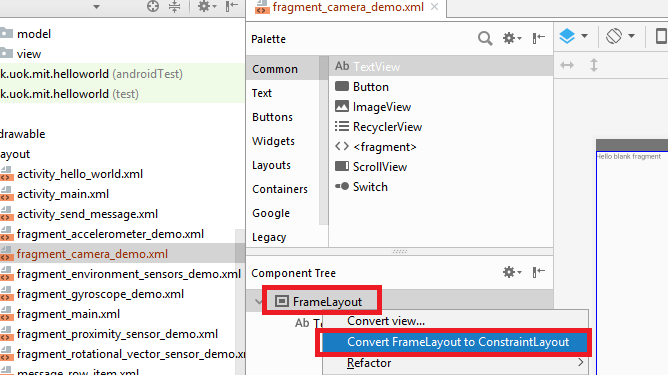
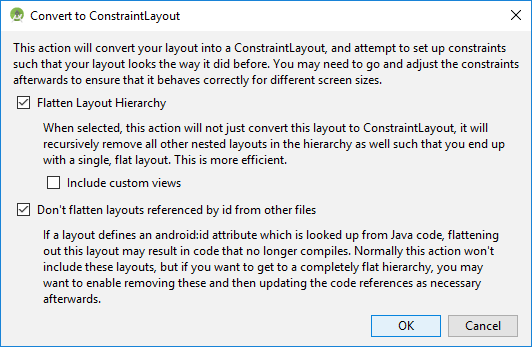
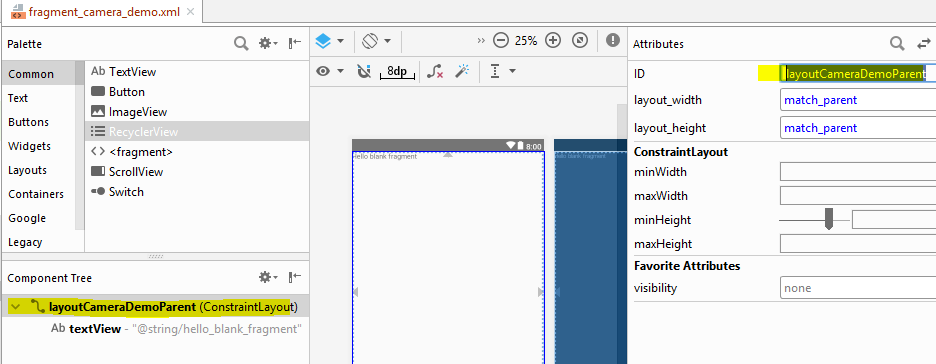
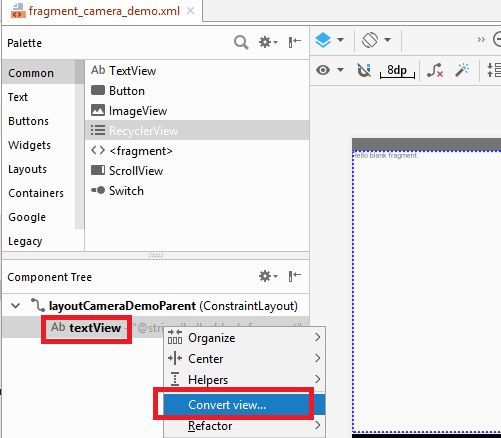
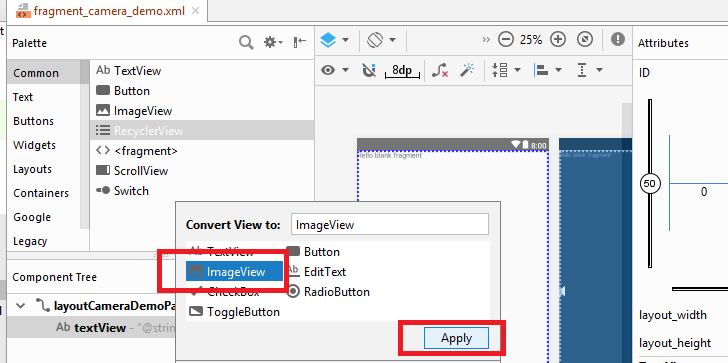
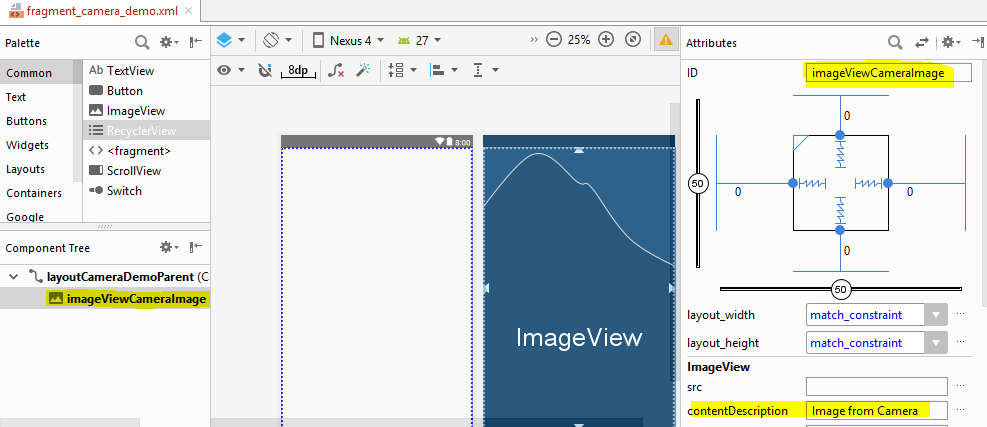
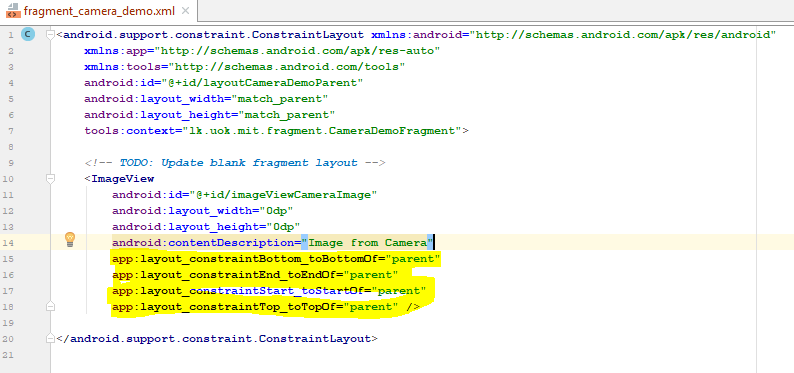
## Constraints overview

* To define a view's position in ConstraintLayout, you must add at least one horizontal and one vertical constraint for the view.
* Each constraint represents a connection or alignment to another view, the parent layout, or an invisible guideline.
* Each constraint defines the view's position along either the vertical or horizontal axis; so each view must have a minimum of one constraint for each axis, but often more are necessary.
* When you drop a view into the Layout Editor, it stays where you leave it even if it has no constraints.
* However, this is only to make editing easier; if a view has no constraints when you run your layout on a device, it is drawn at position [0,0] (the top-left corner).

Lets use a constraint layout in out fragment layout to see how to use a one in an app;

1. Modify the “**fragment\_camera\_demo.xml**” to display the image retrieved from the Camera Intent,

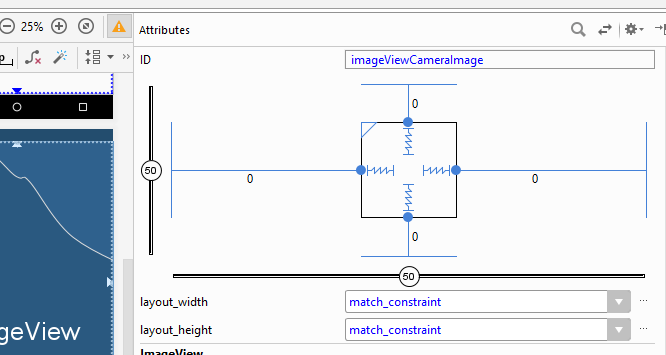
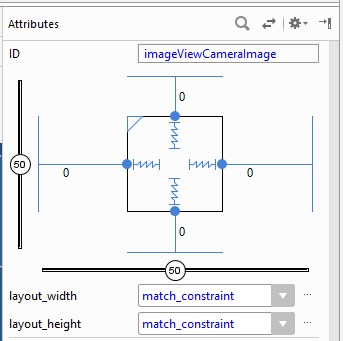
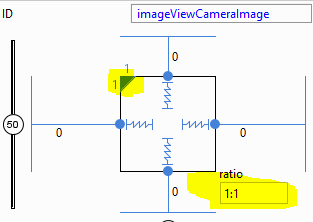
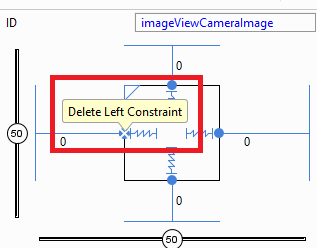
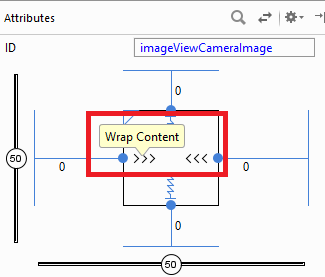
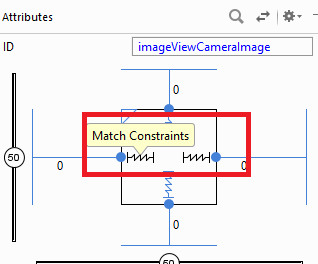
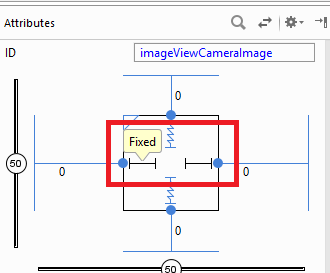
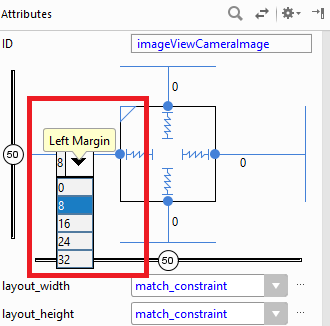
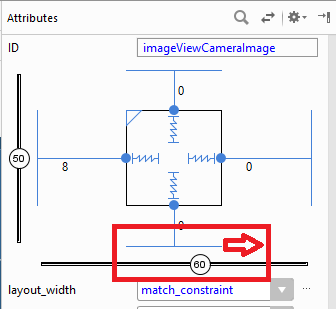
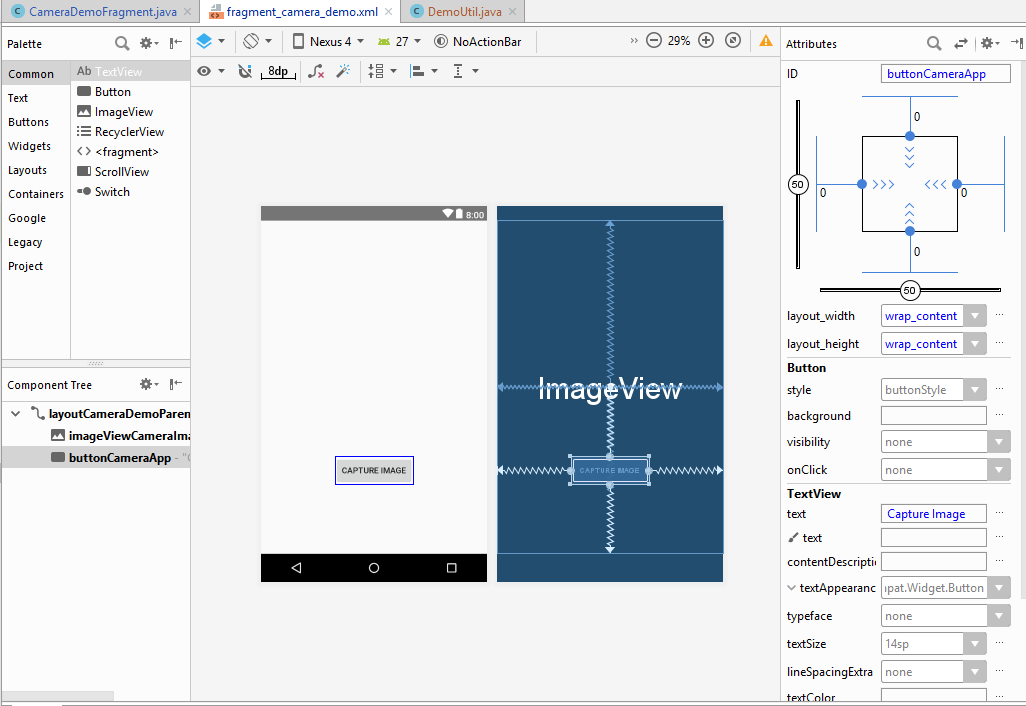
Open the **fragment\_camera\_demo.xml**  file inside “**res/layout**” folder and go to its design view, it should look like below now;  


1. Right click on the “**FrameLayout**” inside the “**Component Tree**” and select “**Convert FrameLayout to ConstraintLayout**”  
     
   1. Then the below confirm box will appear, check both checkboxes and click ok  
      
2. Change the id of the ConstraintLayout to “**layoutCameraDemoParent**” and finally, the component will look like below;  
   
3. Now to convert the “textView” to an ImageView to display the image captured from camera,
   1. Right click on “**textView**” in inside the “**Component Tree**” and select “**Convert View**”  
      
   2. Select “**ImageView**” from popup and change its id to “**imageViewCameraDemo**”  
      
   3. Change the id to “**imageViewCameraImage**” and content description to “**Image from Camera**”  
      
4. Now go to the “Text” view and check, the xml code looks like below;  
   
   1. Note the highlighted code segments, as it was mentioned above, to define a view's position in ConstraintLayout, you must add
      1. Horizontal constraint  
         LEFT:-**app:layout\_constraintStart\_toStartOf="parent"**RIGHT**:-app:layout\_constraintEnd\_toEndOf="parent"**
      2. Vertical constraint  
         TOP:- **app:layout\_constraintTop\_toTopOf="parent"**  
         BOTTOM:- **app:layout\_constraintBottom\_toBottomOf="parent"**  
           
         for the view, and each constraint represents a connection or alignment to **another view**, the **parent layout**, or **an invisible guideline**, and in our case its “PARENT LAYOUT”

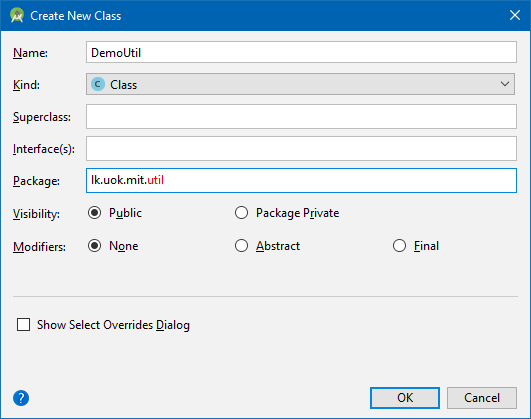
## The view inspector

Near the top of the Attributes window there is a special section called the view inspector.

* It is available only for views in a constraint layout
* It includes controls for several layout attributes, as shown in figure below.

1. Now go again to the “**Design**” view of the “**fragment\_camera\_demo.xml** ” and select “**imageViewCameraImage**” from “**Component Tree**” view and check the “**Attribute window”**;  
   
2. The Attributes window includes controls for
3. size ratio – This is to set the view size to a ratio such as 16:9, if at least one of the view dimensions is set to "match constraints" (0dp).
4. To enable the ratio, click here and then enter the width:height ratio in the input that appears.  
   
5. delete constraint – to remove the constrain  
   
6. height/width mode – to change the way the height and width are calculated, click the symbol to toggle between these settings
7. https://developer.android.com/studio/images/buttons/layout-width-fixed.png - **Fixed**: to specify a specific dimension in the text box below or by resizing the view in the editor
8. https://developer.android.com/studio/images/buttons/layout-width-wrap.png - **Wrap Content**: The view expands only as much as needed to fit its contents.
9. https://developer.android.com/studio/images/buttons/layout-width-match.png - **Match Constraints**: The view expands as much as possible to meet the constraints on each side (after accounting for the view's margins)  
      
   
10. margins – to control the margin for each view  
    
11. constraint bias - the view becomes centered between the two constraints with a bias of 50% by default
12. can adjust the bias by dragging the bias slider in the Attributes window or by dragging the view  
    
13. Add two more button to the layout, and the design view and text view should look like below;
    1. “Design” view of “**fragment\_camera\_demo.xml** ”  
       
    2. Text view of “**fragment\_camera\_demo.xml** ”  
       <**android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"  
        xmlns:app="http://schemas.android.com/apk/res-auto"  
        xmlns:tools="http://schemas.android.com/tools"  
        android:id="@+id/layoutCameraDemoParent"  
        android:layout\_width="match\_parent"  
        android:layout\_height="match\_parent"  
        tools:context="lk.uok.mit.fragment.CameraDemoFragment"**>  
         
        <**ImageView  
        android:id="@+id/imageViewCameraImage"  
        android:layout\_width="0dp"  
        android:layout\_height="wrap\_content"  
        android:contentDescription="Image from Camera"  
        android:scaleType="fitXY"  
        app:layout\_constraintBottom\_toBottomOf="parent"  
        app:layout\_constraintEnd\_toEndOf="parent"  
        app:layout\_constraintStart\_toStartOf="parent"  
        app:layout\_constraintTop\_toTopOf="parent"** />  
         
        <**Button  
        android:id="@+id/buttonCameraApp"  
        android:layout\_width="wrap\_content"  
        android:layout\_height="wrap\_content"  
        android:text="Capture Image"  
        app:layout\_constraintBottom\_toBottomOf="parent"  
        app:layout\_constraintEnd\_toEndOf="parent"  
        app:layout\_constraintStart\_toStartOf="parent"  
        app:layout\_constraintTop\_toBottomOf="@+id/imageViewCameraImage"** />  
         
       </**android.support.constraint.ConstraintLayout**>

After getting familiar with the constrain layout and its specific view inspector, lets get back to our main objective, to display an image captured from camera through the default camera app;

1. Add permission to write the captured image file to external storage, and to access camera by adding below lines to “**AndroidManifest.xml**” file inside “manifests” folder  
   *<!-- Permission to access memory -->*<**uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE"** />  
   *<!-- Permission to access camera -->*<**uses-permission android:name="android.permission.CAMERA"** />
2. Before using the camera, we have to check if a camera is available, if the required permissions are given. In order to write these methods it is possible to use the same “**CameraDemoFragment**” class.   
   But it’s more useful to use a separate utility class to write these methods, since that can be used across many fragments/activities.  
   Let’s create a class called “**DemoUtil**” inside a package named, “**lk.uok.mit.util**”  
   1. Create a new Class, right click on “java” got to “New”-->”Java Class”.
   2. Fill the details as shown below; this class will contain utility methods required for the project  
      
      1. Name:- **DemoUtil**
      2. Kind:- **Class**
      3. Package:- **lk.uok.mit.util**
   3. Next, lets write methods required for the Camera integration, (which can be reused) inside this new class as shown below;

1. Write a common method to check if an app has a set of given user permissions;  
   Open the “**DemoUtil”** class inside “**java**” write below method
   1. Method name:- **hasUserPermissions**
   2. Return Type:- **boolean** - if at least one permission is not given, this will return false
   3. Access modifiers :- **public static**
   4. Parameters:-
      1. **Context context** – the context of which permissions are required
      2. **String[] permissions** – the list of permission to be checked
   5. Logic:-  
      *// Check whether user has all the passed permissions or not.***public static boolean** hasUserPermissions(Context context, String[] permissions) {  
       **boolean** ret = **true**;  
       *//iterate through the permissions passed* **for** (**int** i = 0; i < permissions.**length**; i++) {  
       *// check if the permission is available for each permission* **int** hasPermission = ContextCompat.*checkSelfPermission*(context, permissions[i]);  
       *//if the permission status is not equal to granted, return false* **if** (hasPermission != PackageManager.***PERMISSION\_GRANTED***) {  
       ret = **false**;  
       }  
       }  
       **return** ret;  
      }

1. Write a common method to check if an app has a set of given user permissions;  
   Open the “**DemoUtil”** class inside “**java**” write below method;
   1. Method name:- **isSystemFeatureAvailable**
   2. Return Type:- **boolean** - if the feature is not available, this will return false
   3. Access modifiers :- **public static**
   4. Parameters:-
      1. **Context context** – the context of which permissions are required
      2. **String[] permissions** – the list of permission to be checked
   5. Logic:-  
      *//a method to check if a camera is available in device***public static boolean** isSystemFeatureAvailable(Context context, String featureName) {  
       **if** (context.getPackageManager().hasSystemFeature(featureName)) {  
       *// this device has a camera* **return true**;  
       } **else** {  
       *// no camera on this device* **return false**;  
       }  
      }
2. Write a common method to get a unique file name (to save the image file)  
   Open the “**DemoUtil”** class inside “**java**” write below method;
   1. Method name:- getUniqueFileName
   2. Return Type:- **String** – the generated file name
   3. Access modifiers :- **public static**
   4. Parameters:-
      1. **String prefix** – the prefix of the file, can be used to easily recognize the files
      2. **String fileExtension** – the extension of the file

Logic:-  
**public static** String getUniqueFileName(String prefix, String fileExtension){  
 *//create a simple date format* SimpleDateFormat dateFormat = **new** SimpleDateFormat(**"yyyyMMdd\_HHmmss"**);  
 *// Preparing media file naming convention  
 //get current time* Date currentTime= Calendar.*getInstance*().getTime();  
 *// get the current time using the dateformat object* String timeStamp=dateFormat.format(currentTime);  
 *//construct a unique file name* String fileName = prefix + timeStamp + **"."** + fileExtension;  
 **return** fileName;  
}

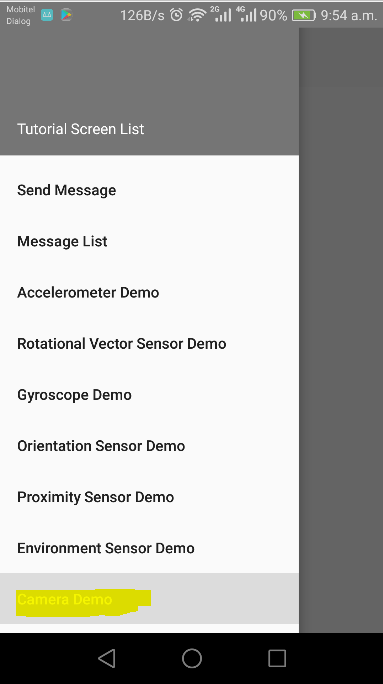
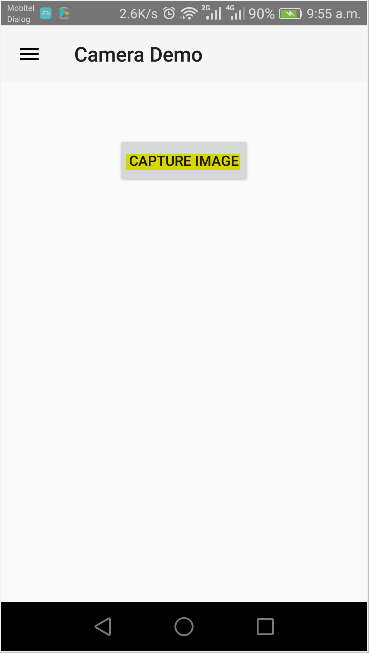
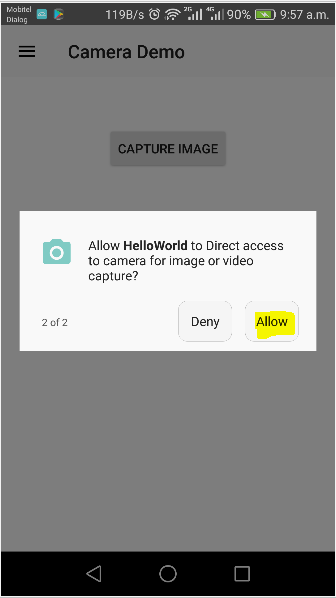
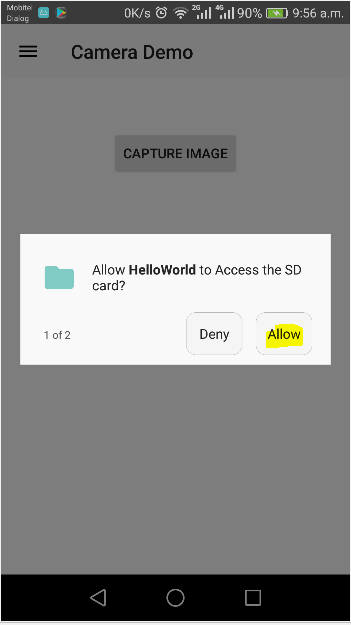
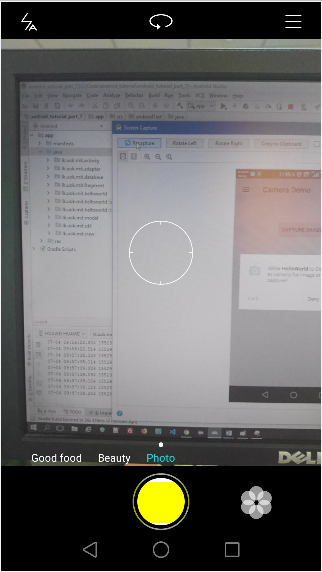
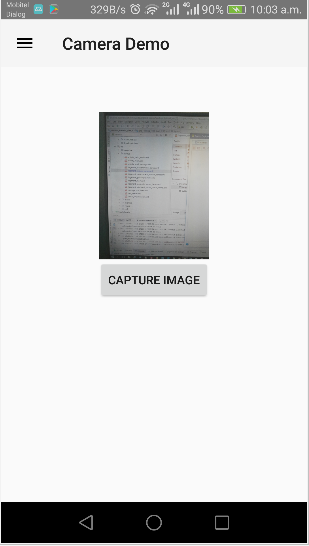
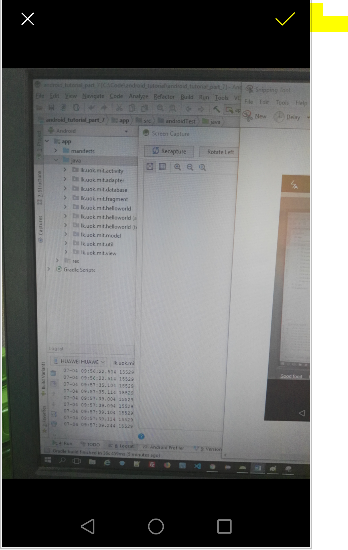
1. Next lets a write a method which will create a file by given name in the given directory, this is to be used to store the captured image data through camera;
   1. Add the below class variables to the “**DemoUtil”** class  
      *//the unique permission request code***public static int** *UNIQUE\_APP\_REQUEST\_CODE* = 1234;  
        
      *//extension of the image file saved***public static final** String ***IMAGE\_EXTENSION*** = **"jpg"**;  
        
      *//the prefix of the image file saved***public static final** String ***IMAGE\_PREFIX*** = **"IMG\_"**;  
        
      *//the physical folder name in which the captured images are saved***public static final** String ***GALLERY\_DIRECTORY\_NAME*** = **"CameraDemo"**;
   2. Method name:- **constructOutputMediaFile**
   3. Return Type:- **File** – the generated file
   4. Access modifiers :- **public static**
   5. Parameters:-
      1. **String fileName** – the name of the generated file
   6. Logic:-  
      */\*\*  
       \* Creates and returns the image or video file before opening the camera  
       \*/***public static** File constructOutputMediaFile(String fileName) {  
       *// External sdcard location* File mediaStorageDir = **new** File(  
       Environment.*getExternalStoragePublicDirectory*(Environment.*DIRECTORY\_PICTURES*),  
       ***GALLERY\_DIRECTORY\_NAME***);  
       *// Create the storage directory if it does not exist* **if** (!mediaStorageDir.exists()) {  
       **if** (!mediaStorageDir.mkdirs()) {  
       Log.*e*(***GALLERY\_DIRECTORY\_NAME***, **"Failed to CREATE create "** + ***GALLERY\_DIRECTORY\_NAME*** + **" directory"**);  
       **return null**;  
       }  
       }  
       *//create a new file* File mediaFile = **new** File(mediaStorageDir.getPath() + File.***separator*** + fileName);  
       Log.*e*(**"GALLERY\_DIRECTORY"**, **"Failed to CREATE create "** + mediaFile);  
       **return** mediaFile;  
      }
2. In order to render the captured image, we need a method to get the URI of the generated file, this can also be written in the “**DemoUtil”** class
   1. Method name:- **getOutputMediaFileUri**
   2. Return Type:- **Uri** – the Uniform Resource Identifier of generated file
   3. Access modifiers :- **public static**
   4. Parameters:-
      1. Context context – the context
      2. File file – The file of which URI is required
   5. Logic:-  
      *//get uri of the media file***public static** Uri getOutputMediaFileUri(Context context, File file) {  
       **return** FileProvider.*getUriForFile*(context, context.getPackageName() + **".provider"**, file);  
      }
3. After capturing image, before displaying in the ImageView, it has to be downsized in order to avoid OutOfMemory exceptions, this is also written in “**DemoUtil”** class
   1. Method name:- **optimizeBitmap**
   2. Return Type:- **Bitmap** – the bitmap of the captured image
   3. Access modifiers :- **public static**
   4. Parameters:-
      1. int sampleSize – the number of pixels in either dimension that correspond to a single pixel in the decoded bitmap
      2. String filePath – The file path of the saved image
   5. Logic:-  
      */\*\*  
       \* Downsizing the bitmap to avoid OutOfMemory exceptions  
       \*/***public static** Bitmap optimizeBitmap(**int** sampleSize, String filePath) {  
       *// bitmap factory* BitmapFactory.Options options = **new** BitmapFactory.Options();  
        
       *// downsizing image as it throws OutOfMemory Exception for larger  
       // images* options.**inSampleSize** = sampleSize;  
        
       **return** BitmapFactory.*decodeFile*(filePath, options);  
      }
4. After saving the image, it should be visible in the gallery of the phone, in order for that to happen in older devices as well, we need to refresh the gallery. Below method is for that purpose and its also written in the “**DemoUtil”** class;
   1. Method name:- **refreshGallery**
   2. Return Type:- **void**
   3. Access modifiers :- **public static**
   4. Parameters:-
      1. **Context context –** the context
      2. **String filePath** – The file path of the saved image
   5. Logic:-  
      */\*\*  
       \* Refreshes gallery on adding new image/video. Gallery won't be refreshed  
       \* on older devices until device is rebooted  
       \*/***public static void** refreshGallery(Context context, String filePath) {  
       *// ScanFile so it will be appeared on Gallery* MediaScannerConnection.*scanFile*(context,  
       **new** String[]{filePath}, **null**,  
       **new** MediaScannerConnection.OnScanCompletedListener() {  
       @Override  
       **public void** onScanCompleted(String path, Uri uri) {  
       }  
       });  
      }
5. In case required to refer to, below are the complete set of imports inside “**DemoUtil”** class  
   **import** android.content.Context;  
   **import** android.content.pm.PackageManager;  
   **import** android.graphics.Bitmap;  
   **import** android.graphics.BitmapFactory;  
   **import** android.media.MediaScannerConnection;  
   **import** android.net.Uri;  
   **import** android.os.Environment;  
   **import** android.support.v4.content.ContextCompat;  
   **import** android.support.v4.content.FileProvider;  
   **import** android.util.Log;  
     
   **import** java.io.File;  
   **import** java.text.SimpleDateFormat;  
   **import** java.util.Calendar;  
   **import** java.util.Date;
6. Now the common methods are completed and lets write code to capture images in “**CameraDemoFragment**”, modify it as shown below
   1. Open the “**CameraDemoFragment**” inside “java”
   2. Modify the “**onCreateView**” method inside “**CameraDemoFragment**” and add the text to display on title bar as shown below;  
      @Override  
      **public** View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {  
       *//set the text appear in title bar* getActivity().setTitle(**"Camera Demo"**);  
       **return** inflater.inflate(R.layout.***fragment\_camera\_demo***, container, **false**);  
      }
   3. Add below class variables to “**CameraDemoFragment**” class  
      *//a code to identify the requested context of the capera app***private static final int *CAMERA\_CAPTURE\_IMAGE\_REQUEST\_CODE*** = 1;  
        
      *// Bitmap sampling size***public static final int *BITMAP\_SAMPLE\_SIZE*** = 8;  
        
      *//to hold the bitmap of the captured image***private** Bitmap **bitmap**;  
        
      *//the image view***private** ImageView **imageView**;  
        
      *//the button to capture image***private** Button **buttonCaptureImage**;  
        
      *//the constructed storage path of the captured image***private** String **imageStoragePath**;  
        
      *//the context***private** Context **context**;  
        
      *//permission array required execute the code here***private** String[] **requiredPermissions** = {Manifest.permission.***WRITE\_EXTERNAL\_STORAGE***, Manifest.permission.***CAMERA***};
7. Write a method inside “**CameraDemoFragment**” class to capture image, that is to start the camera app by passing required data to capture the image and return in to our app;
   1. Method name:- **captureImage**
   2. Return Type:- **void**
   3. Access modifiers :- **private**
   4. Parameters:- None
   5. Logic:-

*/\*\*  
 \* Capturing Camera Image will launch camera app requested image capture  
 \*/***private void** captureImage() {  
 Intent intent = **new** Intent(MediaStore.***ACTION\_IMAGE\_CAPTURE***);  
 *//get a unique name for the image file* String fileName =DemoUtil.*getUniqueFileName*(DemoUtil.***IMAGE\_PREFIX***,DemoUtil.***IMAGE\_EXTENSION***);  
 *//construct a file to store the captured image* File file = DemoUtil.*constructOutputMediaFile*(fileName);  
 **if** (file != **null**) {  
 *//if the file is created and returned, get its path* **imageStoragePath** = file.getAbsolutePath();  
 }  
 *//get the uniform resource locator of the file* Uri fileUri = DemoUtil.*getOutputMediaFileUri*(**context**, file);  
 *//put the URI in to camera intent as data, it will populate this file* intent.putExtra(MediaStore.***EXTRA\_OUTPUT***, fileUri);  
 *// start the image capture Intent* startActivityForResult(intent, ***CAMERA\_CAPTURE\_IMAGE\_REQUEST\_CODE***);  
}

1. Write a method inside “**CameraDemoFragment**” class to generate a bitmap from the captured image file and set to the ImageView
   1. Method name:- **previewCapturedImage**
   2. Return Type:- **void**
   3. Access modifiers :- **private**
   4. Parameters:- None
   5. Logic:-

*/\*\*  
 \* Display image from gallery  
 \*/***private void** previewCapturedImage() {  
 *//construct the bit map ater reducing the file seize* Bitmap bitmap = DemoUtil.*optimizeBitmap*(***BITMAP\_SAMPLE\_SIZE***, **imageStoragePath**);  
 *//set the generated bitmap to the image view* **imageView**.setImageBitmap(bitmap);  
}

1. Inside **CameraDemoFragment** class, override the **onViewCreated** method to initialize the views and add onclick listener to the button  
   @Override  
   **public void** onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {  
    **super**.onViewCreated(view, savedInstanceState);  
    *//initialize the image view* **imageView** = view.findViewById(R.id.***imageViewCameraImage***);  
    *//initialize the button* **buttonCaptureImage** = view.findViewById(R.id.***buttonCameraApp***);  
    *//initalize the context* **context** = getContext();  
    *//set the onclick listener of the button* **buttonCaptureImage**.setOnClickListener(**new** View.OnClickListener() {  
    @Override  
    **public void** onClick(View v) {  
    *//check if user has given the required permissions* **boolean** hasUserPermissions = DemoUtil.*hasUserPermissions*(**context**, **requiredPermissions**);  
    **if** (!(hasUserPermissions)) {  
    *//if read contact permission is not already granted, request permission* requestPermissions(**requiredPermissions**, DemoUtil.*UNIQUE\_APP\_REQUEST\_CODE*);  
    } **else** {  
    *//check if the camera is available in this device* **boolean** cameraAvailable = DemoUtil.*isSystemFeatureAvailable*(**context**, PackageManager.***FEATURE\_CAMERA***);  
    **if** (cameraAvailable) {  
    *//if camera is available, capture the image* captureImage();  
    }**else**{  
    *//else notify the user* Toast.*makeText*(**context**,  
    **"Camera is not Available!"**, Toast.***LENGTH\_SHORT***)  
    .show();  
    }  
    }  
    }  
    });  
   }
   1. As you could notice, when user clicks on “Capture Image” button, user will be redirected to the camera app, we wrote it in “captureImage()” method above
   2. After user captures the image, he will be redirected back to our intent
   3. In order to capture that, method shown below is required.
2. Inside **CameraDemoFragment,** override the **“onActivityResult”** method to get the captured image from camera intent  
   */\*\*  
    \* Activity result method will be called after closing the camera  
    \*/*@Override  
   **public void** onActivityResult(**int** requestCode, **int** resultCode, Intent data) {  
    *// if the result is capturing Image* **if** (requestCode == ***CAMERA\_CAPTURE\_IMAGE\_REQUEST\_CODE***) {  
    **if** (resultCode == Activity.***RESULT\_OK***) {  
    *// Refreshing the gallery* DemoUtil.*refreshGallery*(**context**, **imageStoragePath**);  
    *// successfully captured the image  
    // display it in image view* previewCapturedImage();  
    } **else if** (resultCode == Activity.***RESULT\_CANCELED***) {  
    *// user cancelled Image capture* Toast.*makeText*(**context**,  
    **"User cancelled image capture"**, Toast.***LENGTH\_SHORT***)  
    .show();  
    } **else** {  
    *// failed to capture image* Toast.*makeText*(**context**,  
    **"Sorry! Failed to capture image"**, Toast.***LENGTH\_SHORT***)  
    .show();  
    }  
    }  
   }
3. In case required to refer to, below are the complete set of imports inside **CameraDemoFragment** class  
   **import** android.Manifest;  
   **import** android.app.Activity;  
   **import** android.content.Context;  
   **import** android.content.Intent;  
   **import** android.content.pm.PackageManager;  
   **import** android.graphics.Bitmap;  
   **import** android.net.Uri;  
   **import** android.os.Bundle;  
   **import** android.provider.MediaStore;  
   **import** android.support.annotation.NonNull;  
   **import** android.support.annotation.Nullable;  
   **import** android.support.v4.app.Fragment;  
   **import** android.view.LayoutInflater;  
   **import** android.view.View;  
   **import** android.view.ViewGroup;  
   **import** android.widget.Button;  
   **import** android.widget.ImageView;  
   **import** android.widget.Toast;  
     
   **import** java.io.File;  
     
   **import** lk.uok.mit.helloworld.R;  
   **import** lk.uok.mit.util.DemoUtil;
4. Now save everything and run the app and check, the output should look like below;

# Android's Bluetooth API

Bluetooth is a very popular technology, especially on mobile devices. It's a technology to discover and transfer data between nearby devices. Virtually every modern mobile device has Bluetooth capabilities these days.

The Android platform includes support for the Bluetooth network stack, which allows a device to wirelessly exchange data with other Bluetooth devices. The application framework provides access to the Bluetooth functionality through the Android Bluetooth APIs.

These APIs let applications wirelessly connect to other Bluetooth devices, enabling point-to-point and multipoint wireless features.  
Using the Bluetooth APIs, an Android application can perform the following:

* Scan for other Bluetooth devices
* Query the local Bluetooth adapter for paired Bluetooth devices
* Establish RFCOMM channels
* Connect to other devices through service discovery
* Transfer data to and from other devices
* Manage multiple connections

In this tutorial let’s see how to use the Android Bluetooth APIs to accomplish the four major tasks necessary to communicate using Bluetooth:

1. setting up Bluetooth in app
2. finding devices that are either paired or available in the local area
3. connecting devices
4. transferring data between devices

## How Bluetooth works;

* In order for Bluetooth-enabled devices to transmit data between each other, they must first form a channel of communication using a pairing process.
* One device, a discoverable device, makes itself available for incoming connection requests.
* Another device finds the discoverable device using a service discovery process.
* After the discoverable device accepts the pairing request, the two devices complete a bonding process where they exchange security keys.
* The devices cache these keys for later use.
* After the pairing and bonding processes are complete, the two devices exchange information.
* When the session is complete, the device that initiated the pairing request releases the channel that had linked it to the discoverable device.
* The two devices remain bonded, however, so they can reconnect automatically during a future session as long as they're in range of each other and neither device has removed the bond.

## Components in Android Bluetooth API

The **android.bluetooth** package provides many interfaces and classes to work with Bluetooth such. Some of them are listed below;

* BluetoothAdapter
* BluetoothDevice
* BluetoothSocket
* BluetoothServerSocket
* BluetoothClass
* BluetoothProfile
* BluetoothProfile.ServiceListener
* BluetoothHeadset
* BluetoothA2dp
* BluetoothHealth
* BluetoothHealthCallback
* BluetoothHealthAppConfiguration

### BluetoothAdapter class

By the help of **BluetoothAdapter** class, we can perform fundamental tasks such as initiate device discovery, query a list of paired (bonded) devices, create a **BluetoothServerSocket** instance to listen for connection requests etc.

#### Constants of BluetoothAdapter class

BluetoothAdapter class provides many constants. Most frequently used ones are;

* ACTION\_REQUEST\_ENABLE
* ACTION\_REQUEST\_DISCOVERABLE
* ACTION\_DISCOVERY\_STARTED
* ACTION\_DISCOVERY\_FINISHED

#### Methods of BluetoothAdapter class

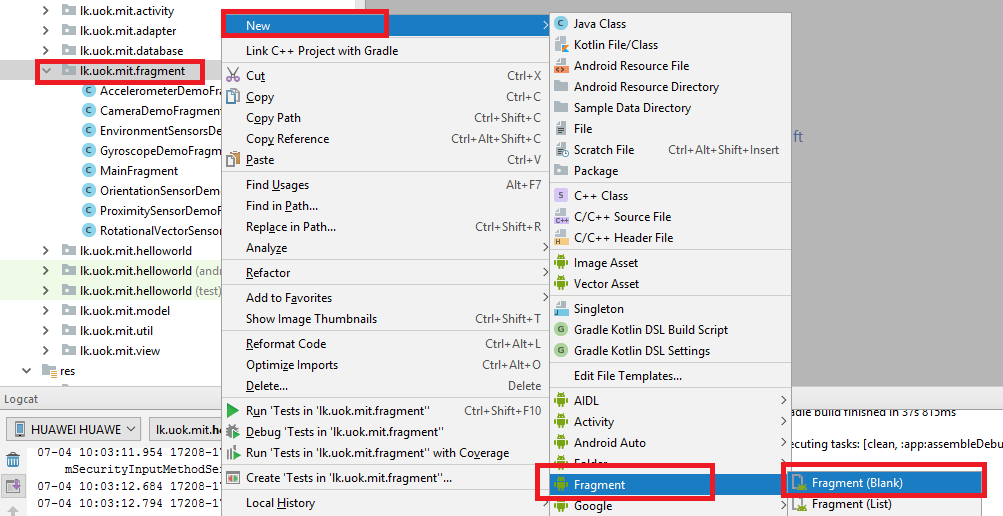
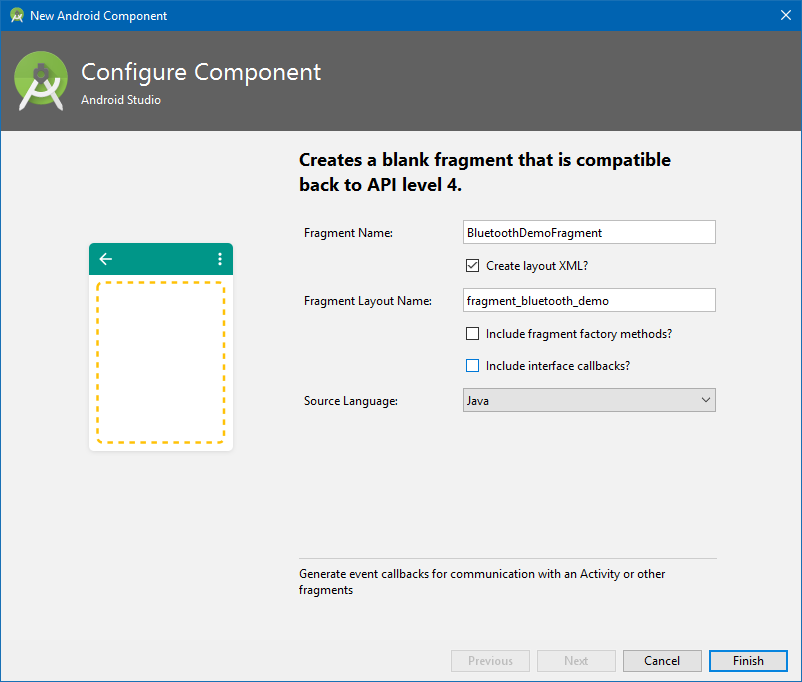
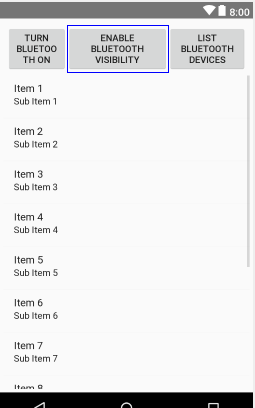
Frequently used methods of **BluetoothAdapter** class are listed below;

* static synchronized BluetoothAdapter getDefaultAdapter()
  + Returns the instance of **BluetoothAdapter**.
* boolean enable()
  + Enables the Bluetooth adapter if it is disabled.
* boolean isEnabled()
  + Returns true if the Bluetooth adapter is enabled.
* boolean disable()
  + Disables the Bluetooth adapter if it is enabled.
* String getName()
  + returns the name of the Bluetooth adapter.
* boolean setName(String name)
  + changes the Bluetooth name.
* int getState()
  + returns the current state of the local Bluetooth adapter.
* Set<BluetoothDevice> getBondedDevices()
  + returns a set of paired (bonded) Bluetooth Device objects.
* boolean startDiscovery()
  + starts the discovery process.

## Bluetooth permissions

In order to use Bluetooth features in an app, it should declare two permissions.

* BLUETOOTH
  + The app needs this permission to perform any Bluetooth communication.
  + E.g.:- requesting a connection, accepting a connection, and transferring data
* ACCESS\_COARSE\_LOCATION or ACCESS\_FINE\_LOCATION
  + A location permission is required because Bluetooth scans can be used to gather information about the location of the user.
  + This information may come from the user's own devices, as well as Bluetooth beacons in use at locations such as shops and transit facilities.
* BLUETOOTH\_ADMIN
  + This permission is required if the app has to initiate device discovery or manipulate Bluetooth settings.
  + Most applications need this permission solely for the ability to discover local Bluetooth devices.
  + The other abilities granted by this permission should not be used, unless the application is a "power manager" that modifies Bluetooth settings upon user request.

1. Add **BLUETOOTH**, **ACCESS\_COARSE\_LOCATION** and **BLUETOOTH\_ADMIN** permissions to the “**AndroidManifest.xml**” of the app
   1. Open the “**AndroidManifest.xml**” inside “**manifests**”
   2. Add permissions as shown below;  
      *<!-- Permission to access bluetooth -->*<**uses-permission android:name="android.permission.BLUETOOTH"** />  
      *<!-- Permission to access bluetooth admin functions -->*<**uses-permission android:name="android.permission.BLUETOOTH\_ADMIN"** />  
      *<!-- Permission to access location -->*<**uses-permission android:name="android.permission.ACCESS\_COARSE\_LOCATION"** />
2. Create a Fragment which demonstrate the usage of Bluetooth
   1. Right click on “**lk.uok.mit.fragment**” and select “**New**”--> “**Fragment**” --> “**Fragment (Blank)**”  
      
   2. Fill the details as shown below;  
      
      1. Fragment Name:- **BluetoothDemoFragment**
      2. Check “Crate layout XML”
      3. Fragment Layout Name:- **fragment\_bluetooth\_demo**
      4. Source Language:- **Java**
3. Add a **menu item** to the **drawer** to access the new “**BluetoothDemoFragment**”
   1. Open the “**drawer\_view.xml**” inside “**res/menu**” folder and add a new menu item with id “**nav\_bluetooth\_demo**” like below;  
      <**item  
       android:id="@+id/nav\_bluetooth\_demo"  
       android:title="Bluetooth Demo"** />
4. Add an entry to the switch case statement inside “**NavigationItemSelectedListener**” inside “**MainActivity**”
   1. Open “**MainActivity**” inside “**java**” folder
   2. Modify the “**setNavigationItemSelectedListene**r(” method of “**NavigationView**” inside “**onCreate**” method like shown below;  
        
      **case** R.id.***nav\_bluetooth\_demo***:  
       *//Open the BluetoothDemoFragment  
       // Begin the transaction* ft = getSupportFragmentManager().beginTransaction();  
       *// Replace the contents of the container with the new fragment* ft.replace(R.id.***fragment\_content***, **new** BluetoothDemoFragment());  
       *// Complete the changes added above* ft.commit();  
       **break**;
5. Modify the “**fragment\_bluetooth\_demo.xml**” file to contain buttons to turn on/off Bluetooth, set visibility of the device and a list view to list Bluetooth devices
   1. Open the “**fragment\_bluetooth\_demo.xml**” file inside “**res/layout**” and add the code below;  
      <**RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"  
       xmlns:tools="http://schemas.android.com/tools"  
       android:layout\_width="match\_parent"  
       android:layout\_height="match\_parent"  
       android:padding="5dip"  
       tools:context="lk.uok.mit.fragment.BluetoothDemoFragment"**>  
        
       <**LinearLayout  
       android:id="@+id/buttonContainer"  
       android:layout\_width="match\_parent"  
       android:layout\_height="wrap\_content"  
       android:padding="5dip"  
       android:orientation="horizontal"**>  
        
       <**Button  
       android:id="@+id/buttonSwitchBluetooth"  
       android:layout\_width="wrap\_content"  
       android:layout\_height="wrap\_content"  
       android:layout\_weight="1"  
       android:text="Turn Bluetooth On"** />  
        
       <**Button  
       android:id="@+id/buttonSwitchBluetoothVisibility"  
       android:layout\_width="wrap\_content"  
       android:layout\_height="wrap\_content"  
       android:layout\_weight="1"  
       android:text="Enable Bluetooth Visibility"** />  
        
       <**Button  
       android:id="@+id/buttonListBluetoothDevices"  
       android:layout\_width="wrap\_content"  
       android:layout\_height="wrap\_content"  
       android:layout\_weight="1"  
       android:text="List Bluetooth Devices"** />  
       </**LinearLayout**>  
        
       <**ListView  
       android:id="@+id/listViewBluetoothDevices"  
       android:layout\_below="@id/buttonContainer"  
       android:layout\_width="match\_parent"  
       android:layout\_height="wrap\_content"  
       android:layout\_centerInParent="true"** />  
        
      </**RelativeLayout**>
   2. The Design view should look like below;  
      

## Enable Bluetooth on a device

First let’s write the functionality for the “**buttonSwitchBluetooth**”, that is to switch on or off the Bluetooth on device;

1. Open “**BluetoothDemoFragment**” inside “java” folder and modify the “**onCreateView**” like below to set the title;  
   *//to refer to context***private** Context **context**;  
     
   @Override  
   **public** View onCreateView(LayoutInflater inflater, ViewGroup container,  
    Bundle savedInstanceState) {  
    *//set the text appear in title bar* getActivity().setTitle(**"Bluetooth Demo"**);  
    *//initilaize context* **this**.**context**=getContext();  
    *// Inflate the layout for this fragment* **return** inflater.inflate(R.layout.***fragment\_bluetooth\_demo***, container, **false**);  
   }
2. Add below class variables to “**BluetoothDemoFragment**”  
   *//code to request Bluetooth***public static int** *BLUETOOTH\_REQUEST\_CODE* = 111;  
     
   *//code to request permissions***public static int** *PERMISSION\_REQUEST\_CODE* = 112;  
     
   *//a class variable to refer to the buttonSwitchBluetooth of the layout,  
   //upon its click we will enable/disable Bluetooth of the device;***private** Button **buttonSwitchBluetooth**;  
     
   *//a reference to the Bluetooth adapter*BluetoothAdapter **mBluetoothAdapter**;  
     
   *//permission array required execute the code here***private** String[] **requiredPermissions** = {  
    Manifest.permission.***BLUETOOTH***,  
    Manifest.permission.***BLUETOOTH\_ADMIN***,  
    Manifest.permission.***ACCESS\_COARSE\_LOCATION***};  
     
   *//to store Bluetooth switch status***private boolean isBluetoothSwitchedOn** = **false**;

1. Write a method to check if Bluetooth facility is available in the device in class “**BluetoothDemoFragment**”;
   1. Method name:- **checkIfBluetoothAvailable**
   2. Return Type:- boolean - returns true if Bluetooth is available, false if not available
   3. Access modifiers :- private
   4. Parameters:- None

Logic:-  
**private boolean** checkIfBluetoothAvailable() {  
 **boolean** isAvailable = **true**;  
  
 *//get the Bluetooth adapter* **this**.**mBluetoothAdapter** = BluetoothAdapter.*getDefaultAdapter*();  
  
 *//check if the bluetooth adapter is null or not* **if** (**this**.**mBluetoothAdapter** == **null**) {  
  
 *// Phone does not support Bluetooth so let the user know and exit.* isAvailable = **false**;  
  
 *//create an alert dialog* AlertDialog alert = **new** AlertDialog.Builder(**context**)  
 .setTitle(**"Not compatible"**)  
 .setMessage(**"Your phone does not support Bluetooth"**)  
 .setPositiveButton(**"Exit"**, **new** DialogInterface.OnClickListener() {  
 **public void** onClick(DialogInterface dialog, **int** which) {  
 System.*exit*(0);  
 }  
 })  
 .setIcon(android.R.drawable.***ic\_dialog\_alert***).create();  
  
 *//show the alert* alert.show();  
 }  
  
 **return** isAvailable;  
}

1. Write a method in class “**BluetoothDemoFragment**” to implement the functionality of “**buttonSwitchBluetooth**”, which is, if Bluetooth of device is enabled, it will disable it, otherwise clicking it will enable Bluetooth
   1. Method name:- **switchBluetooth**
   2. Return Type:- void
   3. Access modifiers :- private
   4. Parameters:- None

Logic:-  
**private void** switchBluetooth() {  
 *//check if user has given the required permissions* **boolean** hasUserPermissions = DemoUtil.*hasUserPermissions*(**context**, **requiredPermissions**);  
 **if** (!(hasUserPermissions)) {  
 *//if read contact permission is not already granted, request permission* requestPermissions(**requiredPermissions**, *PERMISSION\_REQUEST\_CODE*);  
 } **else** {  
 *//check if the Bluetooth is available in this device* **boolean** bluetoothAvailable = checkIfBluetoothAvailable();  
 **if** (bluetoothAvailable) {  
 *//if bluetooth is available, check if it is enabled* **if** (!**this**.**mBluetoothAdapter**.isEnabled()) {  
 *//if Bluetooth is not enabled, first set the status* **isBluetoothSwitchedOn** = **false**;  
 *//try to enable Bluetooth by sending user request* Intent enableBT = **new** Intent(BluetoothAdapter.***ACTION\_REQUEST\_ENABLE***);  
 startActivityForResult(enableBT, *BLUETOOTH\_REQUEST\_CODE*);  
 } **else** {  
 *//if bluetooth is enabled, first set the status* **isBluetoothSwitchedOn** = **true**;  
 *//disable Bluetooth* **this**.**mBluetoothAdapter**.disable();  
 *//change text and background color of the button* **buttonSwitchBluetooth**.setText(**"Turn Bluetooth On"**);  
 **buttonSwitchBluetooth**.setBackgroundColor(Color.***GREEN***);  
 *//notify the user* Toast.*makeText*(**context**,  
 **"Bluetooth is turned off!"**, Toast.***LENGTH\_SHORT***).show();  
 }  
 } **else** {  
 *//else notify the user* Toast.*makeText*(**context**,  
 **"Bluetooth is not Available in this device!"**, Toast.***LENGTH\_SHORT***)  
 .show();  
 }  
 }  
}

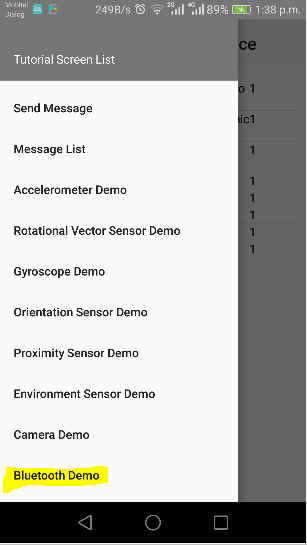
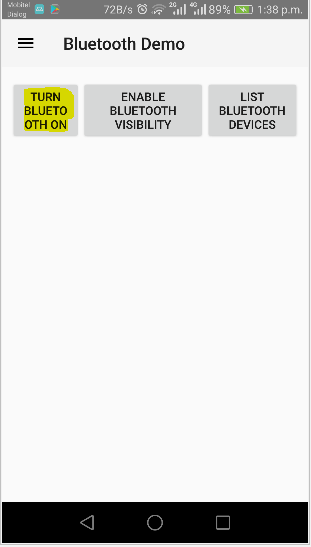
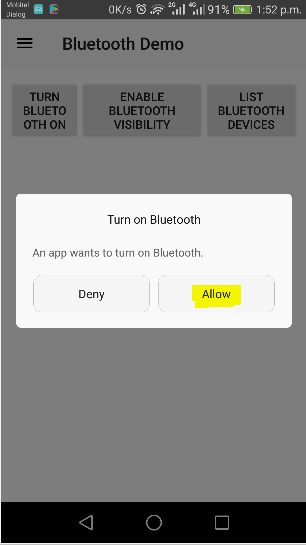
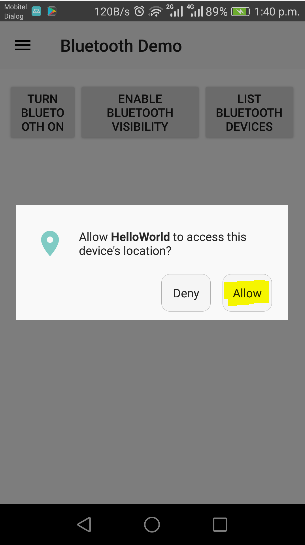
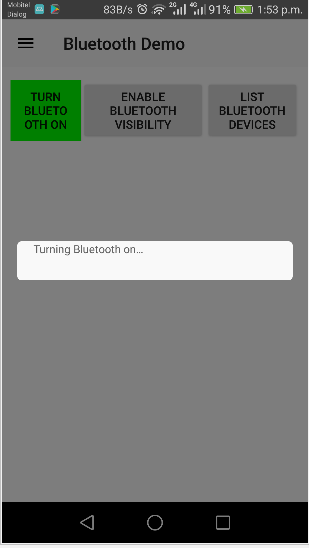
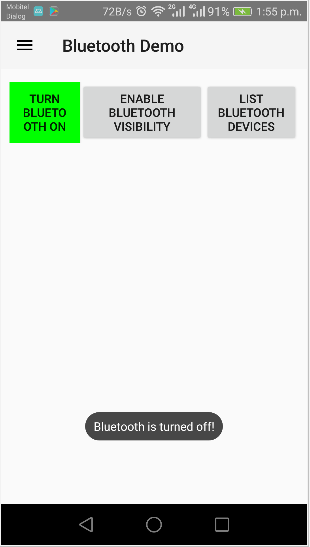
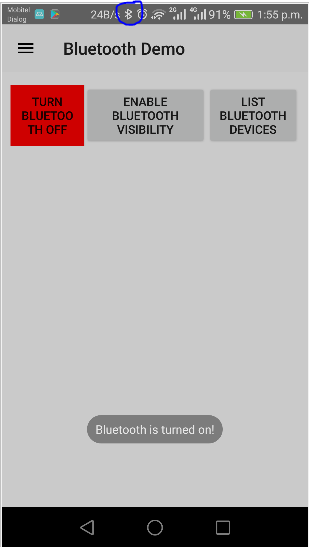
1. Now we have to bind the above method to onClick event of the button “**buttonSwitchBluetooth**”, in order to do that, lets override the “**onViewCreated**” as shown below;  
   @Override  
   **public void** onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {  
    **super**.onViewCreated(view, savedInstanceState);  
    *//get the button reference* **this**.**buttonSwitchBluetooth** = view.findViewById(R.id.***buttonSwitchBluetooth***);  
    *//set the onclick listener of the button* **buttonSwitchBluetooth**.setOnClickListener(**new** View.OnClickListener() {  
    @Override  
    **public void** onClick(View v) {  
    *//call the switchBluetooth method upon click of the button* switchBluetooth();  
    }  
    });  
   }
2. Above code snippets are sufficient to implement the enable/disable of Bluetooth in a device, but for smooth user experience, we have to override “**onActivityResult**” and “**onRequestPermissionsResult**” methods in “**BluetoothDemoFragment**”
   1. Override “**onActivityResult**” –
      1. This is because we used the system activity to ask user to allow enabling Bluetooth on device.
      2. At this point the code execution in our app holds.
      3. Once the user allows or reject, the result will be sent back to our app and our app will continue.
      4. If we do not override this method, there is no way for our app to know the user’s decision.

*/\*\*  
 \* Activity result method will be called after user allowing/denied to enable bluetooth  
 \*/*@Override  
**public void onActivityResult**(**int** requestCode, **int** resultCode, Intent data) {  
  
 *// if the result is activating bluetooth* **if** (requestCode == *BLUETOOTH\_REQUEST\_CODE*) {  
  
 **if** (resultCode == Activity.***RESULT\_OK***) {  
  
 *//notify user* Toast.*makeText*(**context**,  
 **"Bluetooth is turned on!"**, Toast.***LENGTH\_SHORT***)  
 .show();  
  
 *// changing the button text* **buttonSwitchBluetooth**.setText(**"Turn Bluetooth Off"**);  
  
 *//change the background color* **buttonSwitchBluetooth**.setBackgroundColor(Color.***RED***);  
  
 *//changing the status we maintain* **isBluetoothSwitchedOn** = **true**;  
  
 } **else if** (resultCode == Activity.***RESULT\_CANCELED***) {  
  
 *// user cancelled Image capture* Toast.*makeText*(**context**,  
 **"User denied enabling Bluetooth"**, Toast.***LENGTH\_SHORT***)  
 .show();  
 } **else** {  
  
 *// failed to capture image* Toast.*makeText*(**context**,  
 **"Sorry! Failed to enable Bluetooth"**, Toast.***LENGTH\_SHORT***)  
 .show();  
 }  
 }  
}

* 1. Override “**onRequestPermissionsResult**” – The reason to override this is same as above

@Override  
**public void** onRequestPermissionsResult(**int** requestCode, @NonNull String[] permissions, @NonNull **int**[] grantResults) {  
 **super**.onRequestPermissionsResult(requestCode, permissions, grantResults);  
 *// if the result is requesting permissions* **if** (requestCode == *PERMISSION\_REQUEST\_CODE*) {  
 *//declare a variable of type boolean to hold if the user has granted the permission or not* **boolean** hasGranted = **true**;  
 *//first check if the request code matched with our request  
 //check if any results are granted* **int** length = grantResults.**length**;  
 **if** (length > 0) {  
 *//when permissions are available, iterate through them to check if granted* **for** (**int** i = 0; i < length; i++) {  
 **if** (grantResults[i] != PackageManager.***PERMISSION\_GRANTED***) {  
 hasGranted = **false**;  
 }  
 }  
 }  
 *//if granted, show a notification accordingly* **if** (hasGranted) {  
 *//call switch bluetooth method again to execute the logic* switchBluetooth();  
 } **else** {  
 *//if not granted, show a notification accordingly* Toast.*makeText*(**context**, **"User denied permission for Bluetooth"**, Toast.***LENGTH\_LONG***).show();  
 }  
 }  
}

1. For your reference, below are the all imports used in “**BluetoothDemoFragment**” up to now;  
   **import** android.Manifest;  
   **import** android.app.Activity;  
   **import** android.app.AlertDialog;  
   **import** android.bluetooth.BluetoothAdapter;  
   **import** android.content.Context;  
   **import** android.content.DialogInterface;  
   **import** android.content.Intent;  
   **import** android.content.pm.PackageManager;  
   **import** android.graphics.Color;  
   **import** android.os.Bundle;  
   **import** android.support.annotation.NonNull;  
   **import** android.support.annotation.Nullable;  
   **import** android.support.v4.app.Fragment;  
   **import** android.view.LayoutInflater;  
   **import** android.view.View;  
   **import** android.view.ViewGroup;  
   **import** android.widget.Button;  
   **import** android.widget.Toast;  
     
   **import** lk.uok.mit.helloworld.R;  
   **import** lk.uok.mit.util.DemoUtil;
2. Save everything, and run the app and check, the output looks like below;

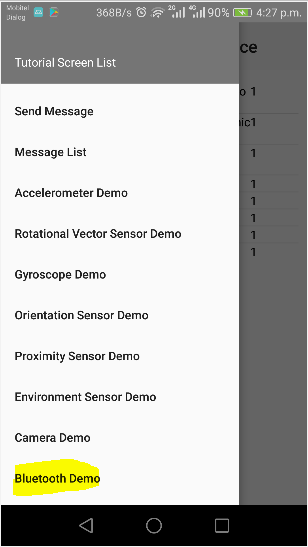
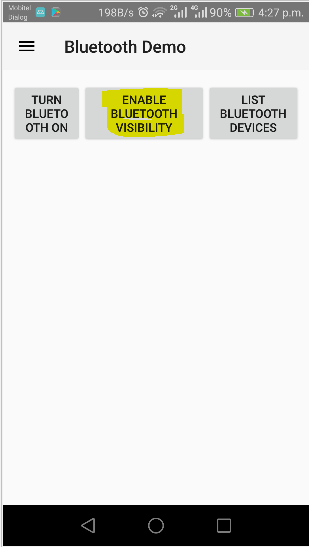
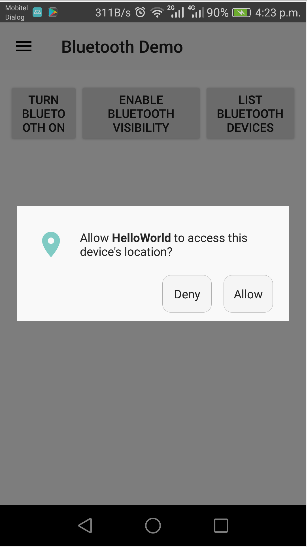
## Enable visibility of device over Bluetooth

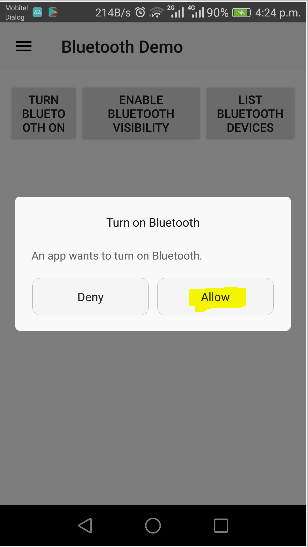
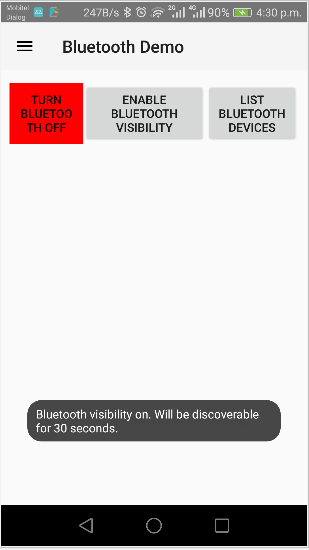
Now let’s write the functionality for the “**buttonSwitchBluetoothVisibility**”, that is to enable or disable the visibility of the device over Bluetooth for other devices

1. Add below two class level variables in to “**BluetoothDemoFragment**”, to refer to “**buttonSwitchBluetoothVisibility**” and to contain the request code for Bluetooth visibility request for user  
   *//code to request bluetooth visibility***public static int** *BLUETOOTH\_VISIBILITY\_REQUEST\_CODE* = 113;  
   *//a class variable to refer to the buttonSwitchBluetooth of the layout,  
   //upon its click we will enable/disable Bluetooth of the device;***private** Button **buttonSwitchBluetoothVisibility**;
2. Write a method in “**BluetoothDemoFragment**” to switch Bluetooth visibility as shown below;
   1. Method name:- **switchBluetoothVisibility**
   2. Return Type:- **void**
   3. Access modifiers :- **private**
   4. Parameters:- **None**

Logic:-  
*//to set visibility of device to other Bluetooth devices***private void** switchBluetoothVisibility() {  
  
 *//check if user has given the required permissions* **boolean** hasUserPermissions = DemoUtil.*hasUserPermissions*(**context**, **requiredPermissions**);  
 **if** (!(hasUserPermissions)) {  
  
 *//if the requested permissions are not already granted, request permission* requestPermissions(**requiredPermissions**, *PERMISSION\_REQUEST\_CODE*);  
 } **else** {  
 *//check if the bluetooth is available in this device* **boolean** bluetoothAvailable = checkIfBluetoothAvailable();  
  
 **if** (bluetoothAvailable) {  
 *//if bluetooth is available, check if it is enabled* **if** (**this**.**mBluetoothAdapter**.isEnabled()) {  
  
 *//check if the device is already visible over bluetooth* **if** (**mBluetoothAdapter**.getScanMode() != BluetoothAdapter.***SCAN\_MODE\_CONNECTABLE\_DISCOVERABLE***) {  
  
 *//if not try to enable Bluetooth visibility by sending user request* Intent getVisible = **new** Intent(BluetoothAdapter.***ACTION\_REQUEST\_DISCOVERABLE***);  
  
 *//make the device visible for 30 seconds, if the user allows* getVisible.putExtra(BluetoothAdapter.***EXTRA\_DISCOVERABLE\_DURATION***, 30);  
  
 *//set the code to request, so we can track the result* startActivityForResult(getVisible, *BLUETOOTH\_VISIBILITY\_REQUEST\_CODE*);  
  
 } **else** {  
 Toast.*makeText*(**context**,  
 **"Device is visible over Bluetooth now!"**, Toast.***LENGTH\_SHORT***).show();  
 }  
 } **else** {  
 *//notify the user that bluetooth of device is turned off* Toast.*makeText*(**context**,  
 **"Bluetooth is turned off!"**, Toast.***LENGTH\_SHORT***).show();  
 }  
 } **else** {  
 *//else notify the user* Toast.*makeText*(**context**,  
 **"Bluetooth is not Available in this device!"**, Toast.***LENGTH\_SHORT***).show();  
 }  
 }  
}

1. Modify the “**onViewCreated**” method inside “**BluetoothDemoFragment**” and call “**switchBluetoothVisibility**” method when user clicks the “**buttonSwitchBluetoothVisibility**”  
   @Override  
   **public void** onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {  
    **super**.onViewCreated(view, savedInstanceState);  
    *//get the button reference* **this**.**buttonSwitchBluetooth** = view.findViewById(R.id.***buttonSwitchBluetooth***);  
    *//set the onclick listener of the button* **buttonSwitchBluetooth**.setOnClickListener(**new** View.OnClickListener() {  
    @Override  
    **public void** onClick(View v) {  
    *//call the switchBluetooth method upon click of the button* switchBluetooth();  
    }  
    });  
     
    *//get the button reference for set visibility button* **this**.**buttonSwitchBluetoothVisibility** = view.findViewById(R.id.***buttonSwitchBluetoothVisibility***);  
    *//set the onclick listener of set visibility button* **buttonSwitchBluetoothVisibility**.setOnClickListener(**new** View.OnClickListener() {  
    @Override  
    **public void** onClick(View v) {  
    *//call the switchBluetooth method upon click of the button* switchBluetoothVisibility();  
    }  
    });  
     
   }
2. Save everything, and run the app and check, the output looks like below;

## List Bluetooth devices

In this step, lets scan for paired Bluetooth devices and display them in the ListView we added in “**fragment\_bluetooth\_demo.xml**”.

In the context of a mobile device, a Bluetooth device can be in either of below three status:-

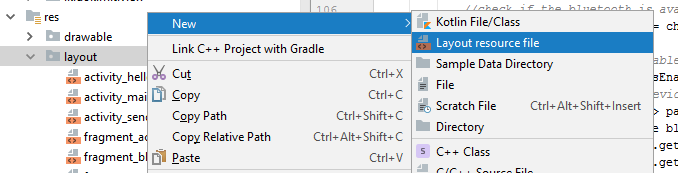
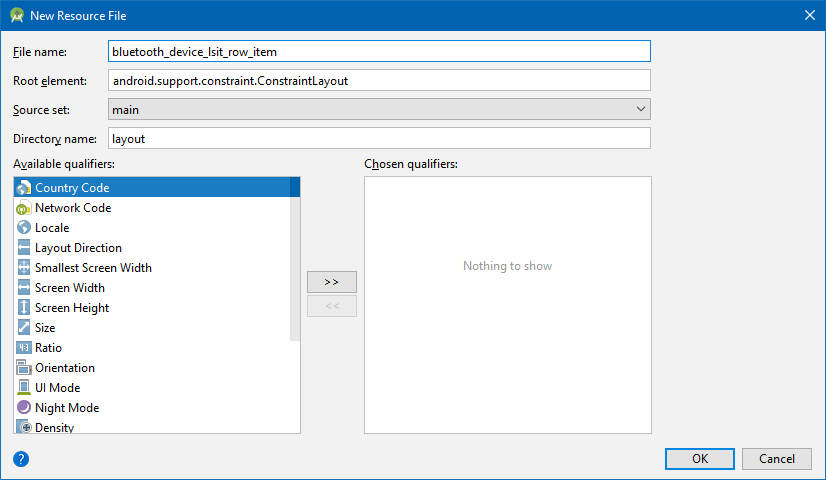
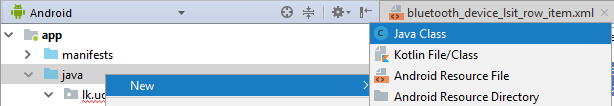
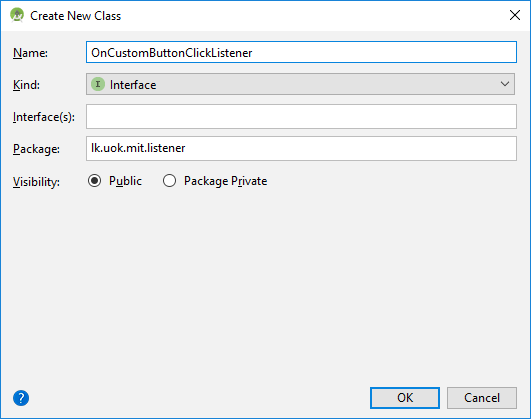
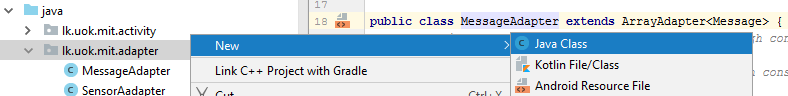
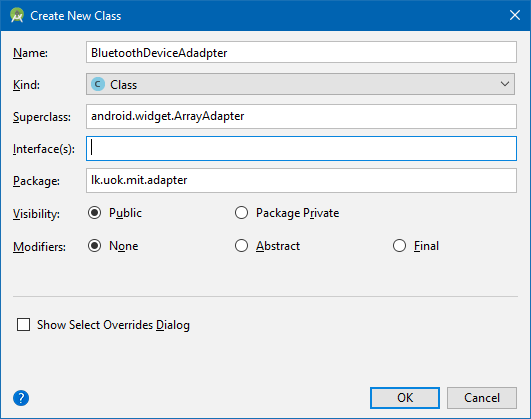
* unknown
* paired
* connected

### Difference between a paired and a connected Bluetooth device

|  |  |
| --- | --- |
| Paired Devices | Connected Devices |
| aware of each other’s existence and **share** a **link key**, which can be used to authenticate, resulting in a connection | **share an RFCOMM channel**, allowing them to send and receive data |
| A device can have many paired devices | A device can have only one connected devices at a given time |

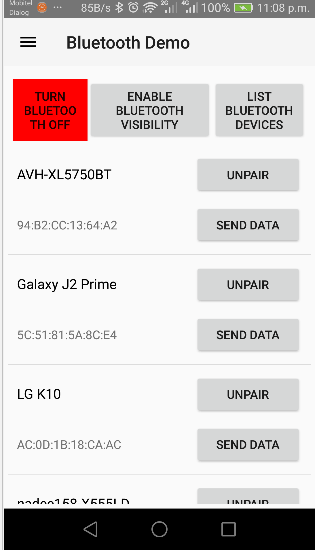
* Bluetooth devices are represented by the **BluetoothDevice** object.
* A list of paired devices can be obtained by invoking the getBondedDevices() method, which returns a set of BluetoothDevice objects.

In order to list the paired Bluetooth devices, we should first create a layout to represent a single row of the list view

1. Create a layout named “**bluetooth\_device\_lsit\_row\_item**.**xml**” inside “**res**” folder, right click on “**res**”, select “**New**”-->”**Layout resource file**”  
   
2. Fill the details as shown below;  
   
   1. File Name:- **bluetooth\_device\_lsit\_row\_item**
   2. Root element:- **RelativeLayout**
   3. Source set:- **main**
   4. Directory Name:- **layout**
3. Open “**bluetooth\_device\_lsit\_row\_item**.**xml**” inside “**res**” folder, got to text view and add below code;  
   *<?***xml version="1.0" encoding="utf-8"***?>*<**RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:padding="10dp"  
    android:layout\_width="match\_parent"  
    android:layout\_height="match\_parent"**>  
     
     
    <**TextView  
    android:id="@+id/textViewDeviceName"  
    android:layout\_width="wrap\_content"  
    android:layout\_height="wrap\_content"  
    android:layout\_toLeftOf="@+id/buttonPairDevice"  
    android:layout\_alignBaseline="@+id/buttonPairDevice"  
    android:layout\_alignParentLeft="true"  
    android:textSize="16sp"  
    android:textColor="#000000"  
    android:text="Galaxy Nexus"** />  
     
    <**TextView  
    android:id="@+id/textViewDeviceAddress"  
    android:layout\_width="wrap\_content"  
    android:layout\_height="wrap\_content"  
    android:layout\_toLeftOf="@+id/buttonSendData"  
    android:layout\_alignParentLeft="true"  
    android:layout\_alignBaseline="@+id/buttonSendData"  
    android:text="000000000"** />  
     
    <**Button  
    android:id="@+id/buttonPairDevice"  
    android:layout\_width="125dp"  
    android:layout\_height="wrap\_content"  
    android:layout\_alignParentTop="true"  
    android:layout\_marginBottom="10dp"  
    android:layout\_alignParentRight="true"  
    android:text="Pair"** />  
     
    <**Button  
    android:id="@+id/buttonSendData"  
    android:layout\_below="@id/buttonPairDevice"  
    android:layout\_width="125dp"  
    android:layout\_height="wrap\_content"  
    android:layout\_alignParentRight="true"  
    android:text="Send Data"** />  
     
   </**RelativeLayout>**  
   1. Here we have added two text views to display device name and address and then a button to pair the device and another button to send data to the device
4. Since the buttons are inside a list view, we have to create a custom click listener to get triggered upon their click events. right click on “**java**” and select “**New**”-->”**Java Class**”  
   
5. Fill in the details as shown below;  
   
   1. Name:- **OnCustomButtonClickListener**
   2. Kind:- **Interface**
   3. Package:- **lk.uok.mit.listener**
6. Open **OnCustomButtonClickListener** inside “java” and code as below  
   **public interface** OnCustomButtonClickListener {  
    **public void** onCustomButtonClick(**int** position);  
   }
7. Next we should create an adapter named “**BluetoothDeviceAdadpter**” to display data, right click on “**lk.uok.mit.adapter**” inside “**java**” and select “**New**”-->”**Java Class**”  
     
   
8. Fill in the details as shown below;  
   
   1. Name:- **BluetoothDeviceAdadpter**
   2. Kind:- **Class**
   3. Superclass:- **android.widget.ArrayAdapter**
   4. Package:- **lk.uok.mit.adapter**
9. Open the “**BluetoothDeviceAdadpter**” inside “**java**” folder and add below class variables;  
   **public class** BluetoothDeviceAdadpter **extends** ArrayAdapter<BluetoothDevice> {  
     
    *//a variable to refer to the passed data set through constructor* **private** List<BluetoothDevice> **bluetoothDevices**;  
     
    *//a variable to refer to the passed context through constructor* **private** Context **context**;  
     
    *//an inflator to inflate layout* **private** LayoutInflater **mInflater**;  
     
    *//a click listener to handle click event of pair/unpair button* **private** OnCustomButtonClickListener **pairButonListener**;  
     
    *//a click listener to handle click event of send data button* **private** OnCustomButtonClickListener **sendDataButonListener**;

1. Add a constructor to match its parent’s constructor as shown below;  
   *//add a constructor to accept context and data set***public** BluetoothDeviceAdadpter(@NonNull Context context, List<BluetoothDevice> bluetoothDevices) {  
    *//call super clase's constructor by passing the context, layout and the data set* **super**(context, R.layout.***bluetooth\_device\_lsit\_row\_item***, bluetoothDevices);  
    **this**.**context** = context;  
    **this**.**bluetoothDevices** = bluetoothDevices;  
    **mInflater** = LayoutInflater.*from*(context);  
   }
2. Add a view holder to hold references to two text boxes and the button we added in “**bluetooth\_device\_lsit\_row\_item.xml**”  
   //the view holder pattern is used to prevent calling findViewById() frequently during the scrolling of ListView  
   //findViewById() is an expensive method in terms of resources**static class** ViewHolder {  
    TextView **textViewDeviceName**;  
    TextView **textViewDeviceAddress**;  
    Button **buttonPairDevice**;  
    Button **buttonSendData**;  
   }
3. Create setter methods to set “**pairButonListener**” and “**sendDataButonListener**” inside “**BluetoothDeviceAdadpter**” as shown below;  
   *//public setter method to set the click listener of pair button***public void** setPairButonListener(OnCustomButtonClickListener pairButonListener) {  
    **this**.**pairButonListener** = pairButonListener;  
   }  
     
   *//public setter method to set the click listener of send data button***public void** setSendDataButonListener(OnCustomButtonClickListener sendDataButonListener) {  
    **this**.**sendDataButonListener** = sendDataButonListener;  
   }
4. Override the “**getView**” method inside “**BluetoothDeviceAdadpter**” to render details from Bluetooth device object  
   @NonNull  
   @Override  
   **public** View getView(**final int** position, @Nullable View convertView, @NonNull ViewGroup parent) {  
    *//declare a variable f type ViewHolder* ViewHolder holder;  
    *//check if the convert view is null;* **if** (convertView == **null**) {  
    *//if the convert is null, it means view is not initilaized yet  
    //initialize the view by inflating the row item layout* convertView = **mInflater**.inflate(R.layout.***bluetooth\_device\_lsit\_row\_item***, **null**);  
    *//initialize the view holder* holder = **new** ViewHolder();  
    *//set the text views and buttons of the view hoder by getting views from layout* holder.**textViewDeviceName** =  
    (TextView) convertView.findViewById(R.id.***textViewDeviceName***);  
    holder.**textViewDeviceAddress** =  
    (TextView) convertView.findViewById(R.id.***textViewDeviceAddress***);  
    holder.**buttonPairDevice** =  
    (Button) convertView.findViewById(R.id.***buttonPairDevice***);  
    holder.**buttonSendData** =  
    (Button) convertView.findViewById(R.id.***buttonSendData***);  
    *//Sets the tag associated with this view* convertView.setTag(holder);  
    } **else** {  
    *//if its not null, means we have already initialized the view holder and set tag once  
    //just get the initialized view holder by tag* holder = (ViewHolder) convertView.getTag();  
    }  
    *//access the bluetooth device in given position from list* BluetoothDevice device = **bluetoothDevices**.get(position);  
    *//set the device name in the text box* holder.**textViewDeviceName**.setText(device.getName());  
    *//set the device address in the text box* holder.**textViewDeviceAddress**.setText(device.getAddress());  
    *//set the button's text based on paired status of the device  
    //if the device is already paired, make the button text to "Unpair"  
    //if the device is NOT paired, make the button text to "Pair"* holder.**buttonPairDevice**.setText((device.getBondState()  
    == BluetoothDevice.***BOND\_BONDED***) ? **"Unpair"** : **"Pair"**);  
    *//set the onclick listner of the pair button as shown below;* holder.**buttonPairDevice**.setOnClickListener(**new** View.OnClickListener() {  
    @Override  
    **public void** onClick(View v) {  
    **if** (**pairButonListener** != **null**) {  
    **pairButonListener**.onCustomButtonClick(position);  
    }  
    }  
    });  
    *//set the onclick listner of the send data button as shown below;* holder.**buttonSendData**.setOnClickListener(**new** View.OnClickListener() {  
    @Override  
    **public void** onClick(View v) {  
    **if** (**sendDataButonListener** != **null**) {  
    **sendDataButonListener**.onCustomButtonClick(position);  
    }  
    }  
    });  
    **return** convertView;  
   }
5. For reference purposes, below are the imports used in **BluetoothDeviceAdadpter** class**;  
   import** android.bluetooth.BluetoothDevice;  
   **import** android.content.Context;  
   **import** android.support.annotation.NonNull;  
   **import** android.support.annotation.Nullable;  
   **import** android.view.LayoutInflater;  
   **import** android.view.View;  
   **import** android.view.ViewGroup;  
   **import** android.widget.ArrayAdapter;  
   **import** android.widget.Button;  
   **import** android.widget.TextView;  
     
   **import** java.util.List;  
     
   **import** lk.uok.mit.helloworld.R;  
   **import** lk.uok.mit.listener.OnCustomButtonClickListener;
6. Now the adapter and the layout for device list view is completed, it’s time to populate the list view inside “**BluetoothDemoFragment**”   
     
   now, open “**BluetoothDemoFragment**” inside “java” folder and add a class variable to refer to the list view in layout, to refer to adapter and device list as below;  
     
   *//a class variable to refer to the buttonListBluetoothDevices of the layout,  
   //upon its click we will get and display the list of bluetooth devices***private** Button **buttonListBluetoothDevices**;  
     
   *//a variable to refer to a list view***private** ListView **listViewBluetoothDevices**;  
     
   *//The adapter used to render list view***private** BluetoothDeviceAdadpter **bluetoothDeviceAdadpter**;  
     
   *//a variable to hold device list***private** ArrayList<BluetoothDevice> **bluetoothDevices**;
7. First lets write a method named “**listBluetoothDevices**” to list the paired devices in the list view of our layout
   1. Method name:- **listBluetoothDevices**
   2. Return Type:- **void**
   3. Access modifiers :- **private**
   4. Parameters:- **None**

Logic:-  
*//to list the Bluetooth devices***private void** listBluetoothDevices() {  
  
 *//check if user has given the required permissions* **boolean** hasUserPermissions = DemoUtil.*hasUserPermissions*(**context**, **requiredPermissions**);  
  
 **if** (!(hasUserPermissions)) {  
 *//if the requested permissions are not already granted, request permission* requestPermissions(**requiredPermissions**, *PERMISSION\_REQUEST\_CODE*);  
  
 } **else** {  
 *//check if the bluetooth is available in this device* **boolean** bluetoothAvailable = checkIfBluetoothAvailable();  
  
 **if** (bluetoothAvailable) {  
 *//if bluetooth is available, check if it is enabled* **if** (**mBluetoothAdapter**.isEnabled()) {  
  
 *//start scanning bluetooth devices* **mBluetoothAdapter**.startDiscovery();  
  
 *//clear the existing items from list* **bluetoothDeviceAdadpter**.clear();  
  
 *//list the currently paired devices* Set<BluetoothDevice> pairedDevices = **mBluetoothAdapter**.getBondedDevices();  
  
 *//iterate over set and add devices to the lsit associated with adapter* **for** (BluetoothDevice device : pairedDevices) {  
 **bluetoothDevices**.add(device);  
 }  
 *//call “notifyDataSetChanged” method of the adapter to reflect the  
 // latest data set available in the “bluetoothDevices”* **bluetoothDeviceAdadpter**.notifyDataSetChanged();  
  
 } **else** {  
 *//notify the user that bluetooth of device is turned off* Toast.*makeText*(**context**,  
 **"Bluetooth is turned off!"**, Toast.***LENGTH\_SHORT***).show();  
 }  
 } **else** {  
 *//else notify the user* Toast.*makeText*(**context**,  
 **"Bluetooth is not Available in this device!"**, Toast.***LENGTH\_SHORT***).show();  
 }  
 }  
}

1. Modify the “**onViewCreated**” method inside “**BluetoothDemoFragment**” and add code shown below (highlighted) to call the method we wrote above and list the paired devise on clicking the “**buttonListBluetoothDevices**” button;  
   @Override  
   **public void** onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {  
    **super**.onViewCreated(view, savedInstanceState);  
     
    *//get the button reference* **buttonSwitchBluetooth** = view.findViewById(R.id.***buttonSwitchBluetooth***);  
     
    *//set the onclick listener of the button* **buttonSwitchBluetooth**.setOnClickListener(**new** View.OnClickListener() {  
    @Override  
    **public void** onClick(View v) {  
    *//call the switchBluetooth method upon click of the button* switchBluetooth();  
    }  
    });  
     
    *//get the button reference for set visibility button* **buttonSwitchBluetoothVisibility** = view.findViewById(R.id.***buttonSwitchBluetoothVisibility***);  
     
    *//set the onclick listener of set visibility button* **buttonSwitchBluetoothVisibility**.setOnClickListener(**new** View.OnClickListener() {  
    @Override  
    **public void** onClick(View v) {  
     
    *//call the switchBluetooth method upon click of the button* switchBluetoothVisibility();  
    }  
    });  
     
    *//initialize the Bluetooth device list* **bluetoothDevices** = **new** ArrayList<BluetoothDevice>();  
     
    *//initialize the ListView* **listViewBluetoothDevices** = view.findViewById(R.id.***listViewBluetoothDevices***);  
     
    *//initialize the adapter* **bluetoothDeviceAdadpter** = **new** BluetoothDeviceAdadpter(**context**, **bluetoothDevices**);  
     
    *//get the button reference for list devices button* **buttonListBluetoothDevices** = view.findViewById(R.id.***buttonListBluetoothDevices***);  
     
    *//set the onclick listener of list devices button* **buttonListBluetoothDevices**.setOnClickListener(**new** View.OnClickListener() {  
    @Override  
    **public void** onClick(View v) {  
    *//call the switchBluetooth method upon click of the button* listBluetoothDevices();  
    }  
    });  
     
    *//set the adapter to the list view* **listViewBluetoothDevices**.setAdapter(**bluetoothDeviceAdadpter**);   
      
   }
2. Now save everything, run the app, and click the button “List Bluetooth devices”, the output should look like below;  
   

## Pair a Bluetooth Device

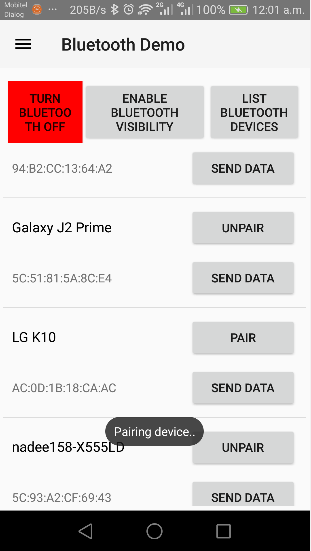
In the device list we have added two buttons for each device listed and using “Pair/Unpair” button, lets write methods to pair or unpair a Bluetooth device.

1. Next, lets write a method to pair a given device inside “**BluetoothDemoFragment”**
   1. Method name:- “**pairDevice**”
   2. Return Type:- **void**
   3. Access modifiers :- **private**
   4. Parameters:- **android.bluetooth.BluetoothDevice**

Logic:-  
**private void** pairDevice(BluetoothDevice device) {  
 **try** {  
 *//use reflection to invoke pairing method named "createBond" on class android.bluetooth.BluetoothDevice* Method method = device.getClass().getMethod(**"createBond"**, (Class[]) **null**);  
 method.invoke(device, (Object[]) **null**);  
 *//use reflection to invoke method named "setPairingConfirmation" on class android.bluetooth.BluetoothDevice  
 //this is to try and stop confirmation from user* device.getClass().getMethod(**"setPairingConfirmation"**, **boolean**.**class**).invoke(device, **true**);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
}

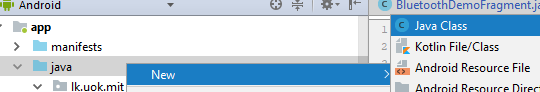
1. Then lets write a method named “**unpairDevice**” to unpair the given device
   1. Method name:- **unpairDevice**
   2. Return Type:- **void**
   3. Access modifiers :- **private**
   4. Parameters:- **android.bluetooth.BluetoothDevice**

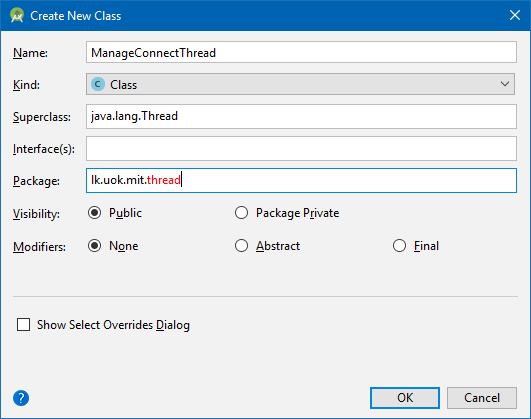
Logic:-  
**private void** unpairDevice(BluetoothDevice device) {  
 **try** {  
 *//use reflection to invoke un-pairing method named "removeBond" on class android.bluetooth.BluetoothDevice* Method method = device.getClass().getMethod(**"removeBond"**, (Class[]) **null**);  
 method.invoke(device, (Object[]) **null**);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
}

1. Note, both above methods used to pair/unpair accepts a Bluetooth device as the argument, this is because the “Pair/Unpair” button exists in our list view, and we should get the device from list on which the action was called.  
     
   Modify the “**onViewCreated**” method inside “**BluetoothDemoFragment**” and add code shown below to bind the “**pairDevice” and** “**unpairDevice**” to the button “**buttonPairDevice**”  
     
   *//set the listener to handle onclick events of each button***bluetoothDeviceAdadpter**.setPairButonListener(**new** OnCustomButtonClickListener() {  
    @Override  
    **public void** onCustomButtonClick(**int** position) {  
     
    *//get the bluetooth device upon the button was clicked using the position passed* BluetoothDevice device = **bluetoothDevices**.get(position);  
     
    **if** (device.getBondState() == BluetoothDevice.***BOND\_BONDED***) {  
    *//if the device is already paired, unpair it* unpairDevice(device);  
     
    } **else** {  
    *//if its not paired, try and pair device* Toast.*makeText*(**context**,  
    **"Pairing device.."**, Toast.***LENGTH\_SHORT***).show();  
    pairDevice(device);  
    }  
    }  
   });
2. Save everything, run the app and check, the output should be like below;  
   

## Send Data to a Bluetooth Device

1. Finally lets write a method to send data to a paired device;   
   For this we need to create a Thread class named “**ManageConnectThread**”, right click on “java”, select “New”->”Java class”



* 1. Fill the details as shown below;  
     
     1. Name:- **ManageConnectThread**
     2. Kind:- **Class**
     3. Superclass:- **java.lang.Thread**
     4. Package:- **lk.uok.mit.thread**
     5. The connecting part, and data sending part to the device is done by using a separate thread, this is because the current execution of the main thread will be kept on hold till connection process tries.
  2. Open the **ManageConnectThread** class and add below class variable and constructor to it;  
       
     **private** BluetoothDevice **device**;  
       
     **public** ManageConnectThread(BluetoothDevice device) {  
      **this**.**device** = device;  
     }
  3. Override the “**run**” method of **ManageConnectThread** and add logic shown below to open a connection to the given device and pass data

@Override  
**public void** run() {  
 DataOutputStream os = **null**;  
 **try** {  
 *// bluetooth serial port service* UUID SERIAL\_UUID = **device**.getUuids()[0].getUuid();  
 **if** (SERIAL\_UUID == **null**) {  
 SERIAL\_UUID = UUID.*fromString*(**"00001101-0000-1000-8000-00805f9b34fb"**);  
 }  
 *// if you don't know the UUID of the bluetooth device service, you can get it like this from android cache* BluetoothSocket clientSocket = **null**;  
 **try** {  
 *//try to get the connection socket by standard method* clientSocket = **device**.createRfcommSocketToServiceRecord(SERIAL\_UUID);  
 } **catch** (Exception e) {  
 Log.*e*(**""**, **"Error creating socket"**);  
 }  
 **try** {  
 *//try to connect to the device* clientSocket.connect();  
 Log.*e*(**""**, **"Connected"**);  
 } **catch** (IOException e) {  
 Log.*e*(**""**, e.getMessage());  
 **try** {  
 Log.*e*(**""**, **"trying fallback..."**);  
 *//if it fails, try to get the connection socket by using reflection* clientSocket = (BluetoothSocket)  
 **device**.getClass().getMethod(**"createRfcommSocket"**,  
 **new** Class[]{**int**.**class**}).invoke(**device**, 1);  
 clientSocket.connect();  
 Log.*e*(**""**, **"Connected"**);  
 } **catch** (Exception e2) {  
 Log.*e*(**""**, **"Couldn't establish Bluetooth connection!"**);  
 }  
 }  
 *//get the output stream from device* os = **new** DataOutputStream(clientSocket.getOutputStream());  
 *//initialize a random byte array* **byte**[] b = **new byte**[20];  
 **new** Random().nextBytes(b);  
 *//write it to the outputstream* os.write(b);  
 *//flush the content* os.flush();  
 *//os.close();* } **catch** (Exception e1) {  
 Log.*e*(**"SENDING DATA"**, e1.getMessage());  
 e1.printStackTrace();  
 } **finally** {  
 **if** (!(os == **null**)) {  
 **try** {  
 os.close();  
 } **catch** (IOException ex) {  
 Log.*e*(**"SENDING DATA"**, ex.getMessage());  
 }  
 }  
 }  
}

1. Write a method to send data to a paired device inside “**BluetoothDemoFragment**”
   1. Method name:- **senDataToDevice**
   2. Return Type:- **void**
   3. Access modifiers :- **private**
   4. Parameters:- **android.bluetooth.BluetoothDevice**
   5. Logic:-  
      **private void** senDataToDevice(BluetoothDevice device) {  
       **try** {  
        
       ManageConnectThread manageConnectThread = **new** ManageConnectThread(device);  
       manageConnectThread.start();  
        
       } **catch** (Exception ex) {  
       Log.*e*(**"senDataToDevice"**, ex.getMessage());  
       }  
        
      }
      1. Here we have created an instance of the thread class passing the selected device and started that thread
      2. Inside the run method of thread class, the code is available to open the connection and send data to the given device
2. Now we should call this “**sendDataToDevice**” method on clicking of “send data” button inside the list view, for that, modify the “**onViewCreated**” method inside “**BluetoothDemoFragment**” and add the code given below;

*//set the listener to handle send data events of each button***bluetoothDeviceAdadpter**.setSendDataButonListener(**new** OnCustomButtonClickListener() {  
 @Override  
 **public void** onCustomButtonClick(**int** position) {  
 BluetoothDevice device = **bluetoothDevices**.get(position);  
 *//check if the device is paired* **if** (device.getBondState() == BluetoothDevice.***BOND\_BONDED***) {  
 Log.*e*(**"SENDING DATA"**, device.getName());  
 senDataToDevice(device);  
 } **else** {  
 Toast.*makeText*(**context**,  
 **"The device is not paired!"**, Toast.***LENGTH\_SHORT***).show();  
 }  
 }  
});

With that, data sending part is completed. Save everything, run the app and check.

## Scan for Bluetooth devices

In order to scan for Bluetooth devices, we should add a **android.content.BroadcastReceiver** to our fragment.Android apps can send or receive broadcast messages from the Android system and other Android apps, similar to the **publish-subscribe design pattern**. These broadcasts are sent when an event of interest occurs. For example, the Android system sends broadcasts when various system events occur, such as when the system boots up or the device starts charging.

In our case, we should get a notification if a Bluetooth device is found. Using a context registered broadcast receiver, lets receive notifications on Bluetooth related events.  
To register a receiver with a context, perform the following steps;

1. Open “**BluetoothDemoFragment**” and create an instance of **BroadcastReceiver,** and override its **onReceived** method inside

**private final** BroadcastReceiver **broadcastReceiver** = **new** BroadcastReceiver() {  
 @Override  
 **public void** onReceive(Context context, Intent intent) {  
 String action = intent.getAction();  
  
 **if** (BluetoothDevice.***ACTION\_BOND\_STATE\_CHANGED***.equals(action)) {  
 **final int** state = intent.getIntExtra(BluetoothDevice.***EXTRA\_BOND\_STATE***, BluetoothDevice.***ERROR***);  
 **final int** prevState = intent.getIntExtra(BluetoothDevice.***EXTRA\_PREVIOUS\_BOND\_STATE***, BluetoothDevice.***ERROR***);  
  
 **if** (state == BluetoothDevice.***BOND\_BONDED*** && prevState == BluetoothDevice.***BOND\_BONDING***) {  
 Toast.*makeText*(context,  
 **"Paired!"**, Toast.***LENGTH\_SHORT***).show();  
 } **else if** (state == BluetoothDevice.***BOND\_NONE*** && prevState == BluetoothDevice.***BOND\_BONDED***) {  
 Toast.*makeText*(context,  
 **"Unpaired!"**, Toast.***LENGTH\_SHORT***).show();  
 }  
  
 **bluetoothDeviceAdadpter**.notifyDataSetChanged();  
 } **else if** (BluetoothAdapter.***ACTION\_DISCOVERY\_STARTED***.equals(action)) {  
 *//discovery starts, we can show progress dialog or perform other tasks* Toast.*makeText*(context,  
 **"Scanning of Bluetooth Devices started!"**, Toast.***LENGTH\_SHORT***).show();  
 } **else if** (BluetoothAdapter.***ACTION\_DISCOVERY\_FINISHED***.equals(action)) {  
 *//discovery finishes, dismis progress dialog* Toast.*makeText*(context,  
 **"Scanning of Bluetooth Devices finished!"**, Toast.***LENGTH\_SHORT***).show();  
 } **else if** (BluetoothDevice.***ACTION\_FOUND***.equals(action)) {  
 *//bluetooth device found* BluetoothDevice device = (BluetoothDevice) intent.getParcelableExtra(BluetoothDevice.***EXTRA\_DEVICE***);  
 **boolean** isDeviceExist = checkIfDeviceExist(device);  
 **if** (!(isDeviceExist)) {  
 **bluetoothDevices**.add(device);  
 **bluetoothDeviceAdadpter**.notifyDataSetChanged();  
 }  
 Toast.*makeText*(context,  
 **"Found device "** + device.getName(), Toast.***LENGTH\_SHORT***).show();  
 }  
 }  
}; **private boolean** checkIfDeviceExist(BluetoothDevice device) {  
 **for** (BluetoothDevice deviceItem : **bluetoothDevices**) {  
 **if** (deviceItem.getAddress().equals(device.getAddress())) {  
 **return true**;  
 }  
 }  
 **return false**;  
}

1. Override the “**onResume**” method of **BluetoothDemoFragment** and create an **IntentFilter** and register the receiver by calling **registerReceiver**(**BroadcastReceiver, IntentFilter**)

@Override  
**public void** onResume() {  
 **super**.onResume();  
 IntentFilter filter = **new** IntentFilter();  
 filter.addAction(BluetoothDevice.***ACTION\_FOUND***);  
 filter.addAction(BluetoothDevice.***ACTION\_BOND\_STATE\_CHANGED***);  
 filter.addAction(BluetoothAdapter.***ACTION\_DISCOVERY\_STARTED***);  
 filter.addAction(BluetoothAdapter.***ACTION\_DISCOVERY\_FINISHED***);  
 getActivity().registerReceiver(**broadcastReceiver**, filter);  
}

This registration will make sure our broadcast receiver will get notifications on finding a Bluetooth device, when pairing state changes, devices visibility starts and ends.

In the broadcast receiver we have defined the actions should take place on each event, most of the time just notifying user.

Override the **onPause** method of **BluetoothDemoFragment** to unsubscribe from broadcast receiver on exit of view

@Override  
**public void** onPause() {  
 **super**.onPause();  
 getActivity().unregisterReceiver(**broadcastReceiver**);  
}

Below are the imports used in **BluetoothDemoFragment**

**import** android.Manifest;  
**import** android.app.Activity;  
**import** android.app.AlertDialog;  
**import** android.bluetooth.BluetoothAdapter;  
**import** android.bluetooth.BluetoothDevice;  
**import** android.content.BroadcastReceiver;  
**import** android.content.Context;  
**import** android.content.DialogInterface;  
**import** android.content.Intent;  
**import** android.content.IntentFilter;  
**import** android.content.pm.PackageManager;  
**import** android.graphics.Color;  
**import** android.os.Bundle;  
**import** android.support.annotation.NonNull;  
**import** android.support.annotation.Nullable;  
**import** android.support.v4.app.Fragment;  
**import** android.util.Log;  
**import** android.view.LayoutInflater;  
**import** android.view.View;  
**import** android.view.ViewGroup;  
**import** android.widget.Button;  
**import** android.widget.ListView;  
**import** android.widget.Toast;  
  
**import** java.lang.reflect.Method;  
**import** java.util.ArrayList;  
**import** java.util.Set;  
  
**import** lk.uok.mit.adapter.BluetoothDeviceAdadpter;  
**import** lk.uok.mit.helloworld.R;  
**import** lk.uok.mit.listener.OnCustomButtonClickListener;  
**import** lk.uok.mit.thread.ManageConnectThread;  
**import** lk.uok.mit.util.DemoUtil;

With this, the Bluetooth API usage demo concludes. Source code for this tutorial part can be found in Git Repository given below: - <https://github.com/nadee158/android_tutorial_part_7.git>

# References

Ableson, F. (2009, June 16). *Tapping into Android's sensors*. Retrieved from www.ibm.com: https://www.ibm.com/developerworks/library/os-android-sensor/index.html

*Android AutoCompleteTextView Example*. (2018, January 1). Retrieved from JavaTPoint: https://www.javatpoint.com/android-autocompletetextview-example

*Application Fundamentals*. (2018, May 2018). Retrieved from Android Developers: https://developer.android.com

*Bluetooth overview*. (2018, Jul 4). Retrieved from developer.android.com: https://developer.android.com/guide/topics/connectivity/bluetooth

CHUGH, A. (2018, April 2). *Android AutoCompleteTextView Example Tutorial*. Retrieved from JournalDev: https://www.journaldev.com/9574/android-autocompletetextview-example-tutorial

*Creating and Using Fragments*. (2015, 01 01). Retrieved from guides.codepath.com: https://guides.codepath.com/android/creating-and-using-fragments

google-developer-training. (2018, June 18). *Working with sensor data*. Retrieved from google-developer-training.gitbooks.io: https://google-developer-training.gitbooks.io/android-developer-advanced-course-practicals/unit-1-expand-the-user-experience/lesson-3-sensors/3-1-p-working-with-sensor-data/3-1-p-working-with-sensor-data.html

Green, P. (2015, April 03). *Using Android Sensors in Your App*. Retrieved from www.sitepoint.com: https://www.sitepoint.com/using-android-sensors-application/

Hathibelagal, A. (2017, January 27). *Android Sensors in Depth: Proximity and Gyroscope*. Retrieved from Envato.com: https://code.tutsplus.com/tutorials/android-sensors-in-depth-proximity-and-gyroscope--cms-28084

Kim, M. (2015, Jul 13). *Create a Bluetooth Scanner With Android's Bluetooth API*. Retrieved from code.tutsplus.com: https://code.tutsplus.com/tutorials/create-a-bluetooth-scanner-with-androids-bluetooth-api--cms-24084

*Sensors Overview*. (2018, June 19). Retrieved from Android Developers: https://developer.android.com/guide/topics/sensors/sensors\_overview

Tutorialspoint. (2018, June 18). *Android - Sensors*. Retrieved from Tutorialspoint: https://www.tutorialspoint.com/android/android\_sensors.htm