|  |
| --- |
| Android Tutorial – Part 5 |

|  |
| --- |
| 6-19-2018 |



Table of Contents

[Introduction 2](#_Toc517642965)

[Create a navigation drawer 3](#_Toc517642966)

[The navigation drawer 3](#_Toc517642967)

[Add a new Main Activity to app 3](#_Toc517642968)

[Add dependencies to Gradle file 5](#_Toc517642969)

[Add a drawer to main activity layout 6](#_Toc517642970)

[Add menu items (links to other activities) to drawer 7](#_Toc517642971)

[Add a header to the nav drawer 9](#_Toc517642972)

[Click events of the menu items 10](#_Toc517642973)

[Add the navigation drawer toggle button to the app bar 12](#_Toc517642974)

[Add the toolbar to layout 12](#_Toc517642975)

[Add the nav drawer toggle button 13](#_Toc517642976)

[Open the drawer when the button is tapped 15](#_Toc517642977)

[Creating a Fragment with a UI 15](#_Toc517642978)

[Activities vs Fragments 16](#_Toc517642979)

[Why a fragment is preferable over an Activity 16](#_Toc517642980)

[Add a Fragment to app 17](#_Toc517642981)

[Sensors in Android 21](#_Toc517642982)

[Sensor types supported by the Android platform. 21](#_Toc517642983)

[Sensor Categories available in Android 22](#_Toc517642984)

[Android Sensor Framework 23](#_Toc517642985)

[SensorManager (**android.hardware.SensorManager**) 23](#_Toc517642986)

[Sensor (**android.hardware.Sensor)** 23](#_Toc517642987)

[SensorEvent (**android.hardware.SensorEvent)** 23](#_Toc517642988)

[SensorEventListener (**android.hardware.SensorEventListener)** 24](#_Toc517642989)

[List the available sensors of Device 24](#_Toc517642990)

[Using the Accelerometer in Android app 29](#_Toc517642991)

[References 37](#_Toc517642992)

# Introduction

This is the part four 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 3),

* Different storage techniques in Android
* SQLite database
* Important classes and their methods of Android’s SQLite package
* Table structure to save a message
* Model class structure to save a message
* SQLiteOpenHelper class, and its implementation
* DB Operation helper class
* Android ListView
* Custom Adapter

<https://github.com/nadee158/android_tutorial_part_4.git>

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

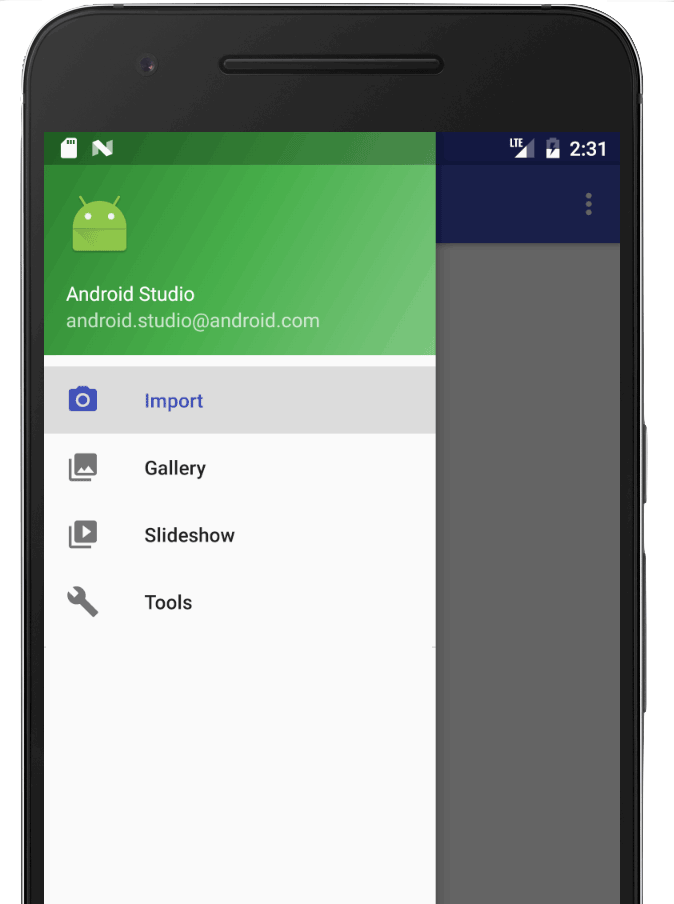
* Navigation Drawer implementation
* Adding dependencies to app’s Gradle build file
* Adding drawer to an activity
* Adding a custom menu resource
* Adding a drawable resource to app
* Changing app themes
* Changing tool bar – adding a toggle button
* FrameLayout in Android
* Fragments in android
* Sensor Framework in Android
* Listing down the sensors available in device
* Using Accelerometer sensor in the app
* Vibrating the device through app

# Create a navigation drawer

In this tutorial, our main focus is to look in to sensors available in android and how to use them for our application. Before starting that part, first we need to create a method to navigate between multiple activities, which will become useful in the next part.

## The navigation drawer

The navigation drawer is a UI panel that shows your app's main navigation menu. It is hidden when not in use, but appears when,

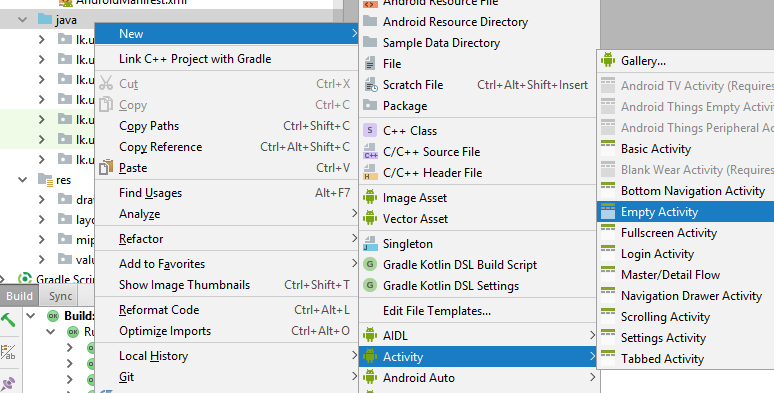
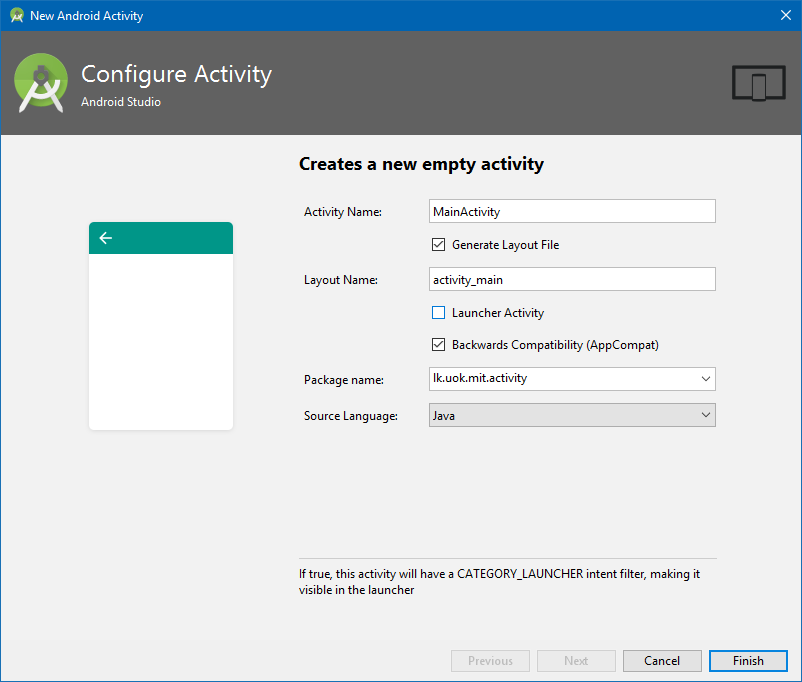
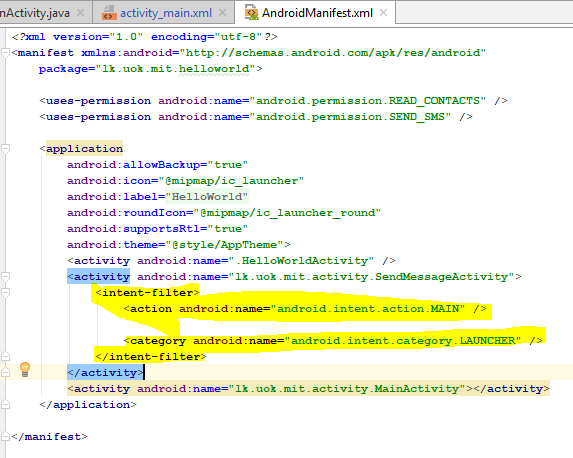
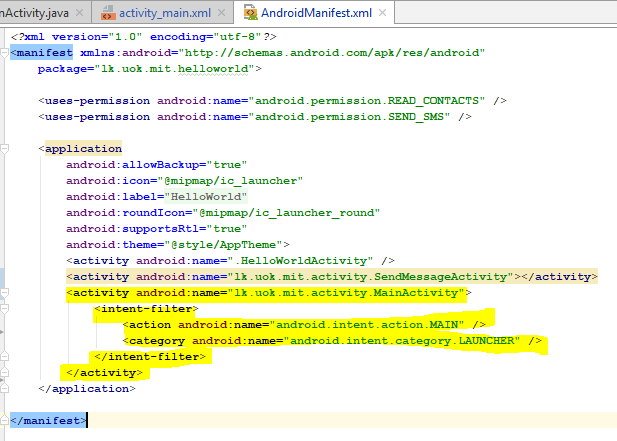
* the user swipes a finger from the left edge of the screen or,
* When at the top level of the app, the user touches the drawer icon in the app bar.  
  
* A navigation drawer is made up of the following components:
  + An instance of the **DrawerLayout** component.
  + An instance of the **NavigationView** component embedded as a child of the **DrawerLayout**.
  + A menu resource file containing the options to be displayed within the navigation drawer.
  + An optional layout resource file containing the content to appear in the header section of the navigation drawer.
  + A listener assigned to the **NavigationView** to detect when an item has been selected by the user.
  + An **ActionBarDrawerToggle** instance to connect and synchronize the navigation drawer to the app bar.
    - The **ActionBarDrawerToggle** also displays the drawer indicator in the app bar which presents the drawer when tapped.

Let’s now see how to implement a navigation drawer using the **DrawerLayout** APIs available in **the Support Library**

## Add a new Main Activity to app

If you could recall, at the time of project creation, we added an activity called “**HelloWorldActivity**”, and it was by default set as the launcher activity (to appear at startup of app).   
After we added the “**SendMessageActivity**”, we made it as the launching activity by modifying the “**AndroidManifest.xml**”.

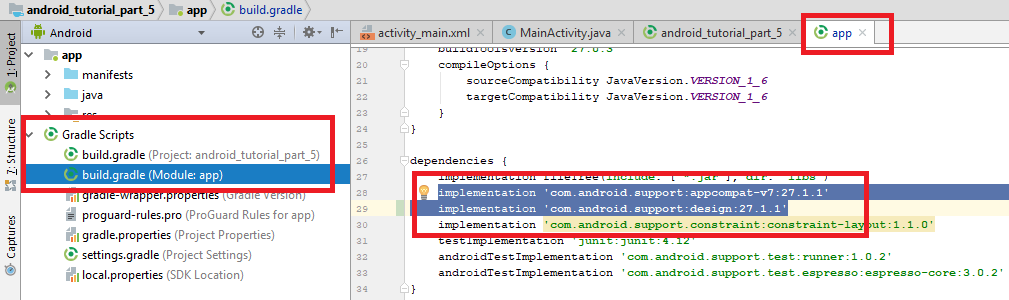
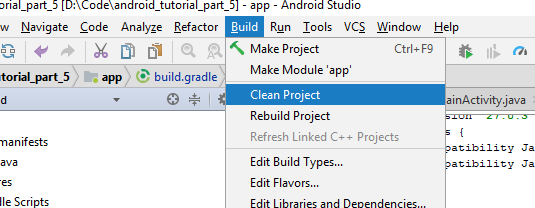
Now we need to add more activities to this app, and each time we will have to keep on making them launching activities. To avoid that, lets add a new activity named “**MainActivity**” and make it the launching activity, and from menu let’s give links to the other activities in the app.

1. Add an empty activity to the app – this is to be used as the landing page of the app  
   Currently, when the app starts up, it directly goes to “**SendMessageActivity**” activity, we are going to change this to a main landing page
   1. Right click on “java”, and go to “**New**”-->”**Activity**”-->”**EmptyActivity**”  
      
   2. Fill the details on the next UI as shown below;  
      
      1. Activity Name:- **MainActivity**
      2. Check the “**Generate Layout File**” option
      3. Layout Name:- **activity\_main**
      4. Package Name:- **lk.uok.mit.activity**
2. Make the “**MainActivity**” as the launcher activity
   1. Open the “**AndroidManifest.xml**” inside “manifests”, which looks like below,  
      
   2. Remove the content within **<intent-filter>** tags from “**SendMessageActivity**” and add it inside “**MainActivity**” as shown below;  
      

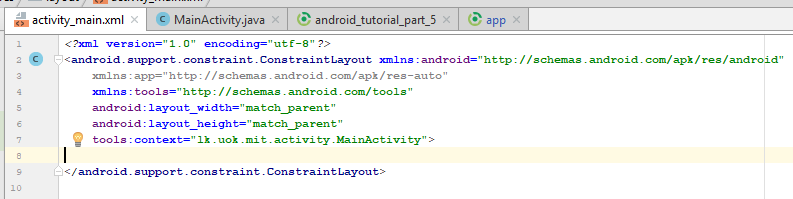
## Add dependencies to Gradle file

**DrawerLayout** acts as a **top-level container** for **window content** that allows for **interactive "drawer"** views to be **pulled out from one or both vertical edges of the window**.

In order to use the **DrawerLayout** and **NavigationView** in the project, it’s required to import the **Design support** and the **Android support** artifacts.

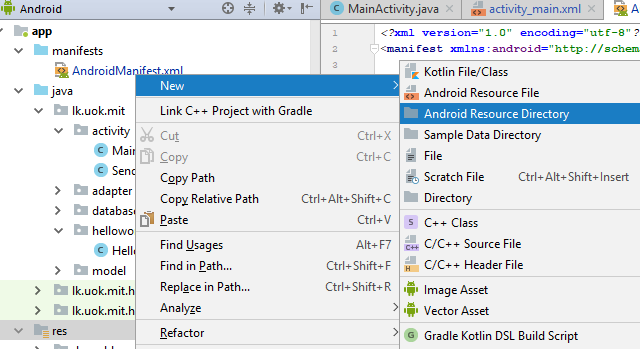
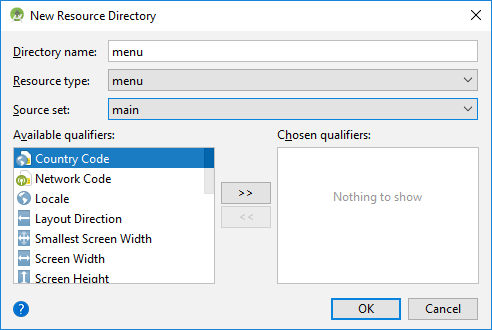
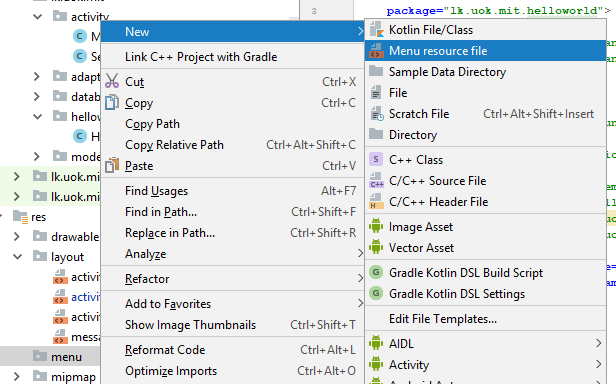
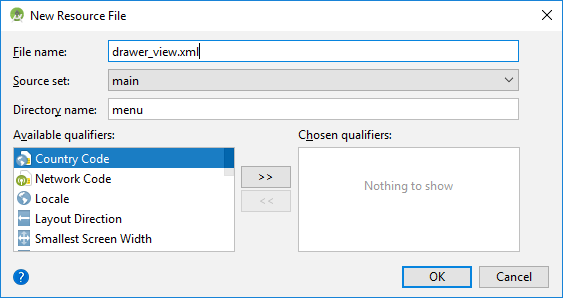
1. Below dependencies should be added to the **build.gradle** file, in order to import them.  
   ***implementation 'com.android.support:appcompat-v7:27.1.1'  
   implementation 'com.android.support:design:27.1.1'***
   1. Open the **build.gradle** file inside **app\build.gradle (**this appears under “Gradle Scripts” section of android studio**)**
   2. Add above two dependencies (if not already available) in “**dependencies{}**” section  
      
   3. After hat **Clean** and **Rebuild** the project, Build🡪Clean / Build🡪Rebuild  
      

## Add a drawer to main activity layout

1. Open “**activity\_main.xml**” created above, inside **res/layout/** and go to “Text” view, it looks like below now;  
     
   1. Above should be changed, and **DrawerLayout** widget and the **NavigationView** should be added like below; (remove all the content from **activity\_main.xml** file and add code given below;)  
      *<?***xml version="1.0" encoding="utf-8"***?>  
      <!-- Use DrawerLayout as root container for activity -->*<**android.support.v4.widget.DrawerLayout xmlns:android="http://schemas.android.com/apk/res/android"  
       xmlns:app="http://schemas.android.com/apk/res-auto"  
       android:id="@+id/drawer\_layout"  
       android:layout\_width="match\_parent"  
       android:layout\_height="match\_parent"  
       android:fitsSystemWindows="true"**>  
        
       *<!-- Layout to contain contents of main body of screen (drawer will slide over this) -->* <**RelativeLayout  
       android:id="@+id/content\_frame"  
       android:layout\_width="match\_parent"  
       android:layout\_height="wrap\_content"**>  
        
       </**RelativeLayout**>  
        
       *<!-- Container for contents of drawer - use NavigationView to make configuration easier -->* <**android.support.design.widget.NavigationView  
       android:id="@+id/nav\_view"  
       android:layout\_width="wrap\_content"  
       android:layout\_height="match\_parent"  
       android:layout\_gravity="start"  
       android:fitsSystemWindows="true"  
       app:menu="@menu/drawer\_view"  
       app:headerLayout="@layout/nav\_header"** />  
        
      </**android.support.v4.widget.DrawerLayout**>  
      1. Here we created a **DrawerLayout** widget with the id **drawer\_layout**.
      2. The above layout uses a **DrawerLayout** with two child views:
         1. **RelativeLayout** - to contain the **main content**
         2. **NavigationView -** for the **contents of the navigation drawer**.
      3. This example demonstrates some important layout characteristics:
         1. The **RelativeLayout** is set to match the parent view's width and height,
            1. Because it represents the entire UI when the navigation drawer is closed.
         2. The **NavigationView** (**the drawer**) must specify its horizontal gravity with the **android:layout\_gravity** attribute.
         3. The **NavigationView** sets **android:fitsSystemWindows** to "**true**" to ensure the contents of the drawer don’t overlay the status bar and other system windows.
         4. The **DrawerLayout** also uses **fitsSystemWindows** as a sign that it needs to inset its children (such as the main content view), but still draw the status bar background in that space as per the material design specs
            1. Which defaults to app theme’s **colorPrimaryDark**.

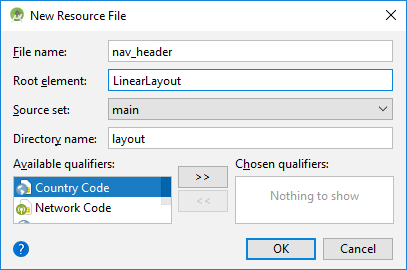
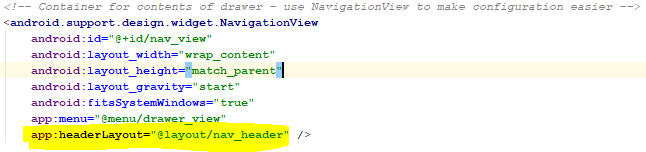
## Add menu items (links to other activities) to drawer

To add menu items, android provides a specific type of resource called “menu”. To configure the menu items listed in the drawer, a menu resource should be specified with the **app:menu** attribute. First a menu resource should be created.

1. To Create a folder named menu under “**res**” folder :-
   1. Right click on “**res**”, select “New”🡪”**Android Resource**” directory   
      
   2. Fill the details in the next screen as shown below;  
      
      1. Directory Name:- menu
      2. Resource Type:- select “menu”
      3. Source set:- main
2. Create the menu resource file with name “**drawer\_view.xml**” inside “**res/menu**” folder
   1. Right click on “**menu**”, select “**New**”🡪 “**Menu Resource File**”  
      
   2. Fill the details in the next screen as shown below;  
      
      1. File Name:- **drawer\_view.xml**
      2. Source Set:- **main**
      3. Directory:- **menu**
   3. Open the **drawer\_view.xml** inside **“res/menu**” folder, and add two menu items to it like below;
      1. **nav\_send\_message –** To access “SendMessageActivity”
      2. **nav\_hello\_world –** To access “HelloWorldActivity”
   4. The whole **drawer\_view.xml** should look like below now;*<?***xml version="1.0" encoding="utf-8"***?>*<**menu xmlns:android="http://schemas.android.com/apk/res/android"**>  
       <**group android:checkableBehavior="single"**>  
       <**item  
       android:id="@+id/nav\_send\_message"  
       android:title="Send Message"** />  
       <**item  
       android:id="@+id/nav\_hello\_world"  
       android:title="Message List"** />  
       </**group**>  
      </**menu**>
   5. Note:- above plain string values are used for demo purposes, please use string resource files each time a string value is used in android to label/give a notification/message etc.
   6. Each menu item can be given an icon if desired using “**android:icon**”
      1. Eg:- ***android:icon="@drawable/ic\_menu\_camera"***
3. Now we created a menu, and it should be added to the “**NavigationView”** of the “**activity\_main.xml**”
   1. Open **activity\_main.xml**  inside res/layout folder
   2. Find the **“<android.support.design.widget.NavigationView**” and add “**drawer\_view**” to its “**app:menu**” attribute like shown below;   
      <**android.support.design.widget.NavigationView  
       android:id="@+id/nav\_view"  
       android:layout\_width="wrap\_content"  
       android:layout\_height="match\_parent"  
       android:layout\_gravity="start"  
       android:fitsSystemWindows="true"  
       app:menu="@menu/drawer\_view"**/>

## Add a header to the nav drawer

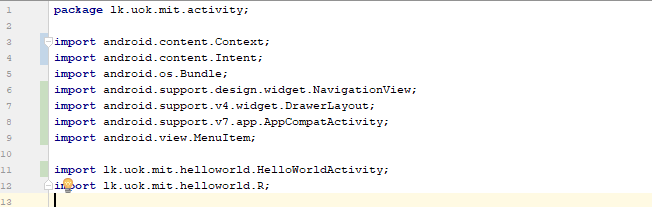
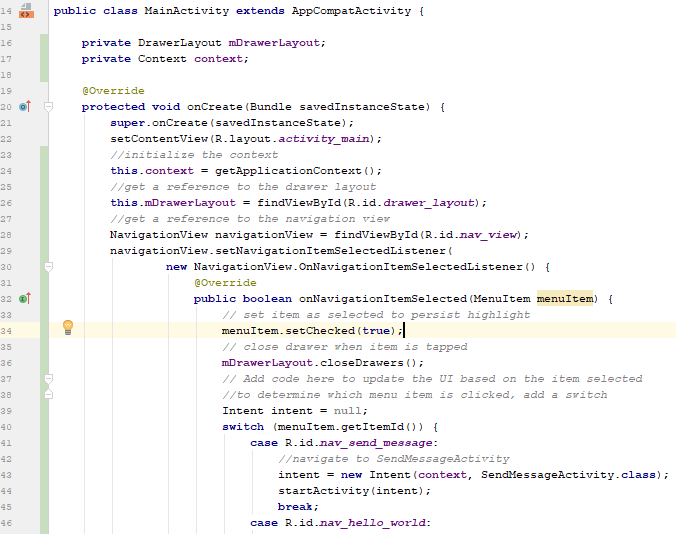
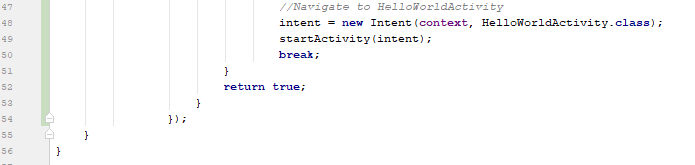
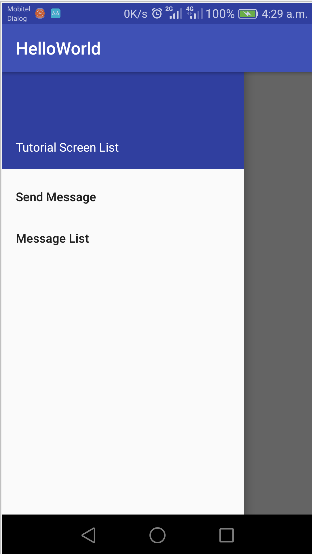
Android provides the option to add a header to the top of the “**drawer**”, that is by allowing to define a separate layout for that using “***app:headerLayout***” attribute of “**NavigationView**”.

1. Define a new layout for the header, named “**nav\_header.xml**” inside “**res/layout**” folder, containing a **LinearLayout** as root element and a **TextView**
   1. Go to “**res/layout**”, right click on it, select “**New**”🡪”**Layout Resource File**”
   2. In the next screen, fill the details as shown below;  
      
      1. File Name:- **nav\_header.xml**
      2. Root element:- **LinearLayout**
      3. Source Set**:- main**
      4. Directory name:- **layout**
2. Open the **nav\_header.xml** inside“**res/layout**” and add below attributes to the LinearLayout tag
   1. **android:layout\_width**="match\_parent"
   2. **android:layout\_height**="192dp"
   3. **android:background**="?attr/colorPrimaryDark"
   4. **android:padding**="16dp"
   5. **android:theme**="@style/ThemeOverlay.AppCompat.Dark"
   6. **android:orientation**="vertical"
   7. **android:gravity**="bottom"
3. Add a text view inside the linear layout, to add he title of the menu bar  
   <**TextView  
    android:layout\_width="match\_parent"  
    android:layout\_height="wrap\_content"  
    android:text="Tutorial Screen List"  
    android:textAppearance="@style/TextAppearance.AppCompat.Body1"**/>
4. The complete code of **nav\_header.xml**  will look like below;  
   *<?***xml version="1.0" encoding="utf-8"***?>*<**LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout\_width="match\_parent"  
    android:layout\_height="192dp"  
    android:background="?attr/colorPrimaryDark"  
    android:padding="16dp"  
    android:theme="@style/ThemeOverlay.AppCompat.Dark"  
    android:orientation="vertical"  
    android:gravity="bottom"**>  
    <**TextView  
    android:layout\_width="match\_parent"  
    android:layout\_height="wrap\_content"  
    android:text="Tutorial Screen List"  
    android:textAppearance="@style/TextAppearance.AppCompat.Body1"**/>  
   </**LinearLayout**>
5. Now add “**app:headerLayout**” to the “**NavigationView**” of the **activity\_main.xml** like shown below;  
   

## Click events of the menu items

Now the UI design part of the “**drawer**” is completed. Now let’s add the code to “**MainActivity.java**”, to handle the taps or clicks on menu items and navigate user to the desired view.

To receive callbacks when the user taps a list item in the drawer, implement the **OnNavigationItemSelectedListener** interface and attach it to the **NavigationView** by calling **setNavigationItemSelectedListener()**

1. Open the “java/MainActivity”, and add the below code inside “onCreate” method  
   *//initialize the context***this**.**context** = getApplicationContext();  
   *//get a reference to the drawer layout***this**.**mDrawerLayout** = findViewById(R.id.***drawer\_layout***);  
   *//get a reference to the navigation view*NavigationView navigationView = findViewById(R.id.***nav\_view***);  
   *//set the navigation item selected listen to get the click item*navigationView.setNavigationItemSelectedListener(  
    *//create an anonymous inner class, and override* ***onNavigationItemSelected*** method  
    **new** NavigationView.OnNavigationItemSelectedListener() {  
    @Override  
    **public boolean** onNavigationItemSelected(MenuItem menuItem) {  
    *// set item as selected to persist highlight* menuItem.setChecked(**true**);  
    *// close drawer when item is tapped* **mDrawerLayout**.closeDrawers();  
    *// Add code here to update the UI based on the item selected   
    //to determine which menu item is clicked, add a switch* Intent intent = **null**;  
    **switch** (menuItem.getItemId()) {  
    **case** R.id.***nav\_send\_message***:  
    *//navigate to SendMessageActivity* intent = **new** Intent(**context**, SendMessageActivity.**class**);  
    startActivity(intent);  
    **break**;  
    **case** R.id.***nav\_hello\_world***:  
    *//Navigate to HelloWorldActivity* intent = **new** Intent(**context**, HelloWorldActivity.**class**);  
    startActivity(intent);  
    **break**;  
    }  
    **return true**;  
    }  
    });
2. The complete code looks like below  
     
     
   
3. Now a working **drawer** is available in the app, save everything, clean and run the app and check;  
   To view drawer, swipe across the screen from left to right; it will appear like below;  
   

## Add the navigation drawer toggle button to the app bar

At this point, we have implemented a working navigation drawer. The **DrawerLayout** provides built-in support for users to **open** and **close** the **navigation drawer** with a **swipe** on the side of the screen.

But if the UI design includes an app bar, app should also allow users to open and close the drawer by touching the drawer icon on the top left of the app bar as shown below.



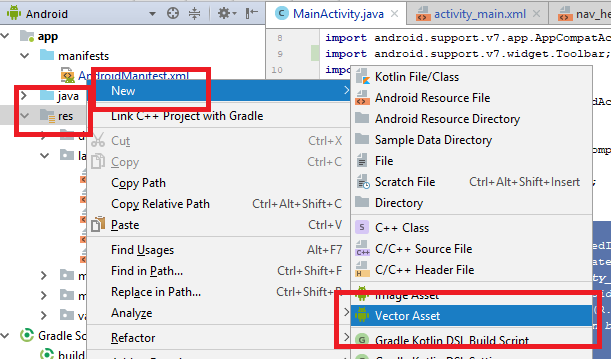
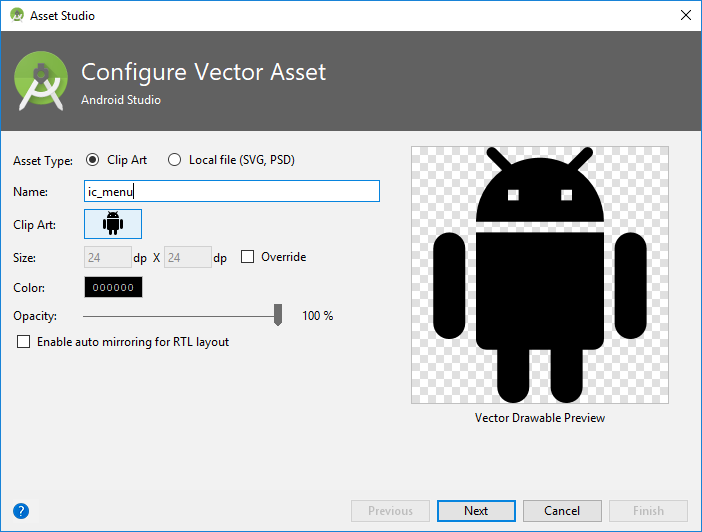
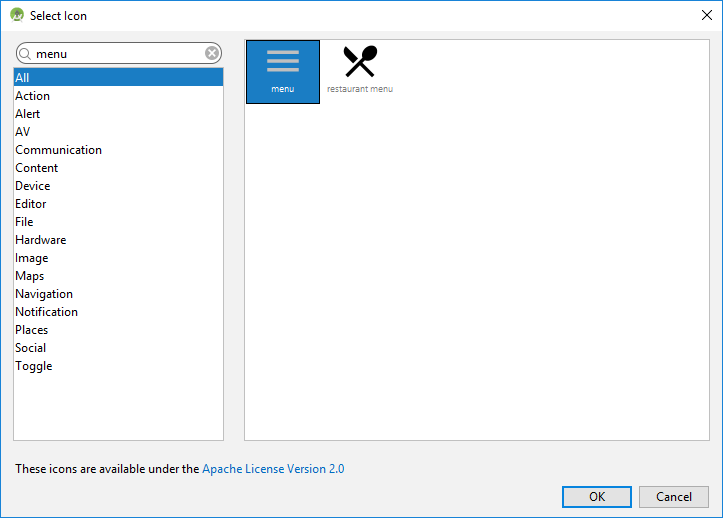
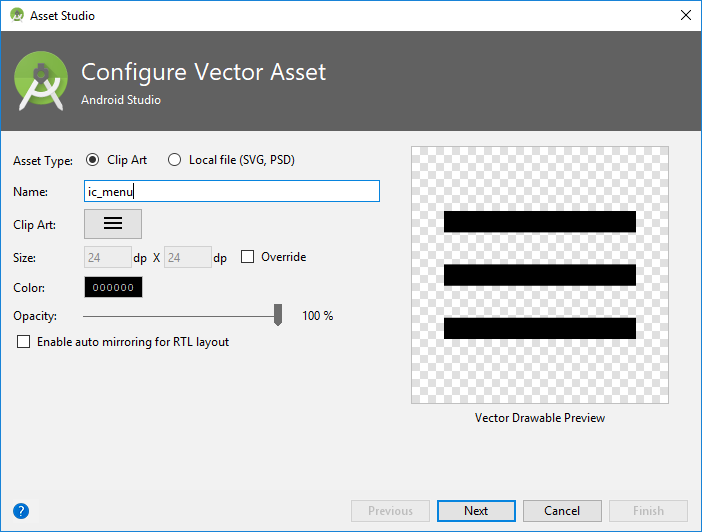
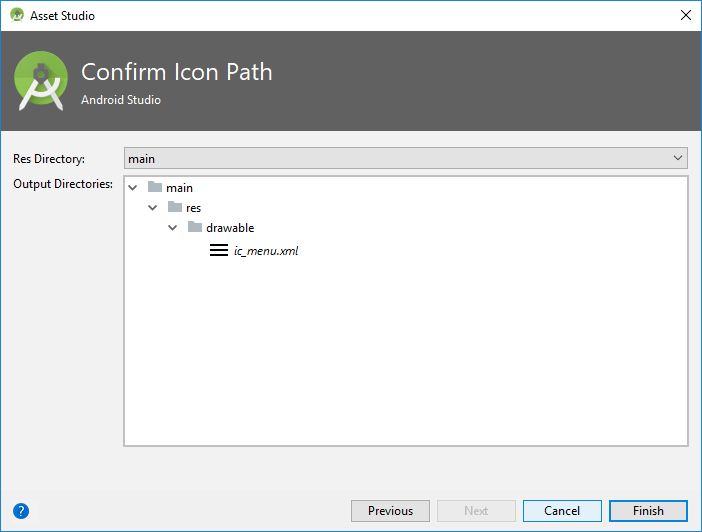
Lets implement a toggle button to the app bar of our app;

### Add the toolbar to layout

1. Open the “**activity\_main.xml**” inside “**res/layout**”, go to the “**Text**” view and add a toolbar in to the “**RelativeLayout**” as shown below;  
   *<!-- Layout to contain contents of main body of screen (drawer will slide over this) -->*<**RelativeLayout  
    android:id="@+id/content\_frame"  
    android:layout\_width="match\_parent"  
    android:layout\_height="wrap\_content"**>  
     
    <**android.support.v7.widget.Toolbar  
    android:id="@+id/toolbar"  
    android:layout\_width="match\_parent"  
    android:layout\_height="?attr/actionBarSize"  
    android:background="?attr/colorPrimary"  
    android:theme="@style/ThemeOverlay.AppCompat.ActionBar"** />  
     
    *<!-- Layout to contain contents of main body of screen (drawer will slide over this) -->* <**FrameLayout  
    android:layout\_below="@id/toolbar"  
    android:id="@+id/fragment\_content"  
    android:layout\_marginTop="5dp"  
    android:layout\_width="match\_parent"  
    android:layout\_height="wrap\_content"**>  
     
    </**FrameLayout**>  
     
   </**RelativeLayout**>
2. Set the app theme to one without the action bar, to **Theme.AppCompat.Light.NoActionBar**
   1. Open “**AndroidManifest.xml**” file inside “**manifests**” and change the app from **“@style/AppTheme**” theme to **“@style/Theme.AppCompat.Light.NoActionBar”** as shown below;  
      <**application  
       android:allowBackup="true"  
       android:icon="@mipmap/ic\_launcher"  
       android:label="@string/app\_name"  
       android:roundIcon="@mipmap/ic\_launcher\_round"  
       android:supportsRtl="true"  
       android:theme="@style/Theme.AppCompat.Light.NoActionBar"**>  
       <**activity android:name=".HelloWorldActivity"** />  
       <**activity android:name="lk.uok.mit.activity.SendMessageActivity"**></**activity**>  
       <**activity android:name="lk.uok.mit.activity.MainActivity"**>  
       <**intent-filter**>  
       <**action android:name="android.intent.action.MAIN"** />  
       <**category android:name="android.intent.category.LAUNCHER"** />  
       </**intent-filter**>  
       </**activity**>  
      </**application**>
3. After removing the action bar, set the added tool bar as the action bar in the MainActivity.java
   1. Open the “**MainActivity**” inside “**java**” folder and set the tool bar as shown below, inside “**onCreate**” method   
      @Override  
      **protected void** onCreate(Bundle savedInstanceState) {  
       **super**.onCreate(savedInstanceState);  
       setContentView(R.layout.***activity\_main***);  
       *//get a reference to the "android.support.v7.widget.Toolbar"* Toolbar toolbar = findViewById(R.id.***toolbar***);  
       *//set the toolbar as the action bar* setSupportActionBar(toolbar);  
       *//initialize the context* **this**.**context** = getApplicationContext();  
       *//get a reference to the drawer layout* **this**.**mDrawerLayout** = findViewById(R.id.***drawer\_layout***);

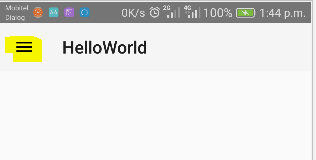
### Add the nav drawer toggle button

Now we should add the button icon that opens the navigation drawer.

1. Add the menu button's icon to the project by importing the icon from Android Studio;
   1. In the Project window, right-click the **res** folder and select **New > Vector Asset**.  
      
   2. Select “**Clip Art”** as the asset type and then click the **“Clip Art”** to open the Select Icon window.  
      
      1. Asset Type:- Select “**Clip Art**”
      2. Name:- “**ic\_menu**”
      3. Clip Art:- Click on **Clip Art** icon
   3. After clicking on the “Clip Art” icon, below screen will appear  
      
      1. Search for "**menu**" and select the **menu icon** (the icon is 3 horizontal lines).
      2. Click OK, and then rename the file to "**ic\_menu**" and click Next to import it.  
         
      3. The next screen will look like below, and the selected resource will be inside **drawable resource** folder  
         
2. Enable the app bar's "**home**" button by calling **setDisplayHomeAsUpEnabled(true)**, and then change it to use the menu icon by calling **setHomeAsUpIndicator(int)**, as shown here;
   1. Open “MainActivity” inside “java” and inside “onCreate” method;  
      @Override  
      **protected void** onCreate(Bundle savedInstanceState) {  
       **super**.onCreate(savedInstanceState);  
       setContentView(R.layout.***activity\_main***);  
       *//get a reference to the "android.support.v7.widget.Toolbar"* Toolbar toolbar = findViewById(R.id.***toolbar***);  
       *//set the toolbar as the action bar* setSupportActionBar(toolbar);  
       *//get a reference to "android.support.v7.app.ActionBar"* ActionBar actionbar = getSupportActionBar();  
       *//Enable the app bar's "home" button* actionbar.setDisplayHomeAsUpEnabled(**true**);  
       *//set the icon we added above as the button* actionbar.setHomeAsUpIndicator(R.drawable.***ic\_menu***);  
       *//initialize the context* **this**.**context** = getApplicationContext();

### Open the drawer when the button is tapped

When clicking on the button added above, the “**drawer**” should open, add the code to add that functionality

1. Open “**MainActivity**” inside “**java**” and inside override the “**onOptionsItemSelected**” method as shown below;  
   @Override  
   **public boolean** onOptionsItemSelected(MenuItem item) {  
    **switch** (item.getItemId()) {  
    **case** android.R.id.***home***:  
    **mDrawerLayout**.openDrawer(GravityCompat.***START***);  
    **return true**;  
    }  
    **return super**.onOptionsItemSelected(item);  
   }
   1. As shown in the highlighted sections above, when the “home” item is selected, the drawer is opened
2. Now run the app and check; the menu button will appear as shown below, click on it and the drawer view will open.  
   
3. As you could notice, the content of the home screen of our app is empty.

# Creating a Fragment with a UI

A **Fragment** is a **self-contained component** with **its own user interface (UI)** and **lifecycle** that can be reused in different parts of an app's UI.   
A Fragment can also be used without a UI, in order to retain values across configuration changes.

A Fragment can be a static part of the UI of an Activity, which means that the Fragment remains on the screen during the entire lifecycle of the Activity, or it can be dynamically added too.

* A fragment has its own layout and its own behavior with its own lifecycle callbacks.
* Can add or remove fragments in an activity while the activity is running.
* Can combine multiple fragments in a single activity to build a multi-pane UI.
* A fragment can be used in multiple activities.
* The fragment life cycle is closely related to the lifecycle of its host activity.
* When the activity is paused, all the fragments available in the activity will also be stopped.
* A fragment can implement a behavior that has no user interface component.
* Fragments were added to the Android API in Android 3 (Honeycomb) with API version 11.

## Activities vs Fragments

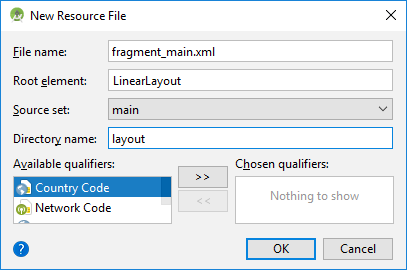
|  |  |
| --- | --- |
| Activities | Fragments |
| are navigation controllers | are content controllers |
| Navigation to other activities through intents. | Layouts and views displaying relevant app content. |
| Presenting navigational components such as the navigation drawer or the viewpager. | Event handling logic associated with relevant views. |
| Hiding and showing relevant fragments using the fragment manager. | View state management logic such as visibility or error handling. |
| Receiving data from intents and passing data between fragments. | Triggering of network request through a client object. |
|  | Retrieval and storage of data from persistence through model objects. |

## Why a fragment is preferable over an Activity

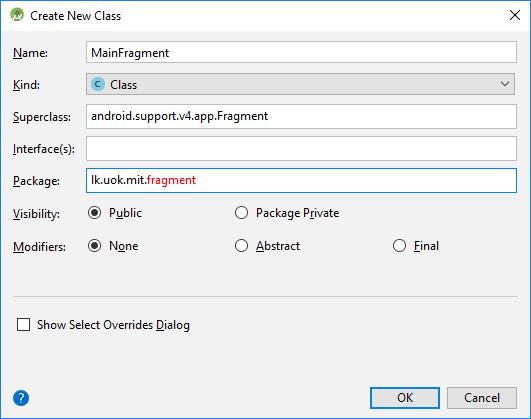
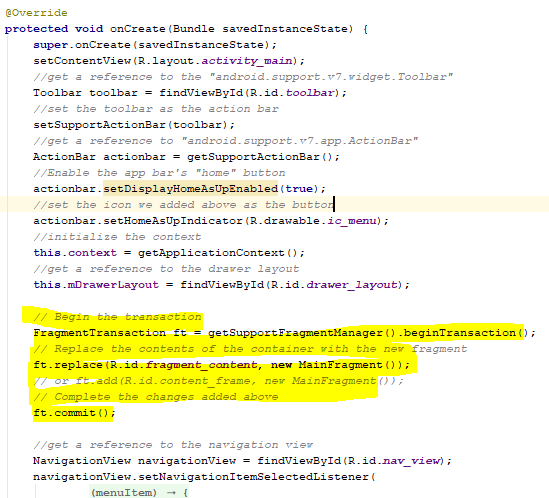
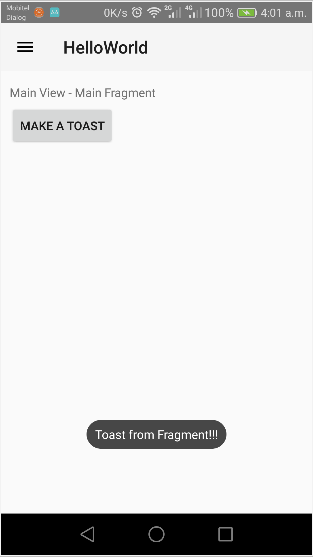
* Activity is expensive.
  + In Fragment, views and property states are separated - whenever a fragment is in backstack, its views will be destroyed.
  + Can stack much more Fragments than Activity.
* Backstack manipulation.
  + With **FragmentManager**, it's easy to clear all the Fragments, insert more than on Fragments and etcs.
  + For Activity, it will be much harder to manage the backstack.
* A much predictable lifecycle.
  + As long as the host Activity is not recycled, the Fragments in the backstack will not be recycled.

## Add a Fragment to app

A fragment, like an activity, has an **XML layout file** and a **Java class** that represents the **Fragment controller**.  
The XML layout file is just like any other layout file, and can be named **fragment\_foo.xml**.   
**Think of a fragment as a partial (re-usable) activity.**

1. Create the XML layout file named “**fragment\_main.xml**”
   1. Right click on “**res/layout**” folder, select “**New**”🡪”**Resource Layout File**”
   2. Fill the details as shown below;  
      
      1. File Name:- **fragment\_main.xml**
      2. Root element:- **LinearLayout**
      3. Source set:- **main**
      4. Directory Name:- **layout**
2. Add a TextView with id “**textViewTitle**” and a button with id “**buttonToast**” to the fragment layout
   1. Open “**fragment\_main.xml**” file inside **res/layout,** and go to “Text” view and add the content below;  
      *<?***xml version="1.0" encoding="utf-8"***?>*<**LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
       android:orientation="vertical"  
       android:padding="10dp"  
       android:layout\_width="match\_parent"  
       android:layout\_height="match\_parent"**>  
        
       <**TextView  
       android:id="@+id/textViewTitle"  
       android:layout\_width="wrap\_content"  
       android:layout\_height="wrap\_content"  
       android:text="Main View - Main Fragment"** />  
        
       <**Button  
       android:id="@+id/buttonToast"  
       android:layout\_marginTop="5dp"  
       android:layout\_width="wrap\_content"  
       android:layout\_height="wrap\_content"  
       android:text="Make a Toast"** />  
        
      </**LinearLayout**>

1. Add a **FrameLayout** to the activity\_main.xml as shown below;
   1. Open the **activity\_main.xml** inside “res/layout” folder
   2. Add a **FrameLayout** inside the **RelativeLayout** as shown below with id **“fragment\_content”***<!-- Layout to contain contents of main body of screen (drawer will slide over this) -->*<**RelativeLayout  
       android:id="@+id/content\_frame"  
       android:layout\_width="match\_parent"  
       android:layout\_height="wrap\_content"**>  
        
       *<!-- Toolbar to contain contents of action bar -->* <**android.support.v7.widget.Toolbar  
       android:id="@+id/toolbar"  
       android:layout\_width="match\_parent"  
       android:layout\_height="?attr/actionBarSize"  
       android:background="?attr/colorPrimary"  
       android:theme="@style/ThemeOverlay.AppCompat.ActionBar"** />  
        
       *<!-- Frame Layout to contain contents of fragments dynamically switched -->* <**FrameLayout  
       android:layout\_below="@+id/toolbar"  
       android:id="@+id/fragment\_content"  
       android:layout\_marginTop="5dp"  
       android:layout\_width="wrap\_content"  
       android:layout\_height="wrap\_content"**>  
        
       </**FrameLayout**>  
        
      </**RelativeLayout**>

1. Add the Java controller for the **Fragment**  named “**MainFragment.java**”
   1. Right click on “java” folder, go to “New”🡪”Java Class”
   2. Fill the details as shown below;  
      
      1. Name:- **MainFragment**
      2. Kind:- **Class**
      3. Superclass:- **android.support.v4.app.Fragment**
      4. Package:- **lk.uok.mit.fragment**
   3. Override the **“onCreateView”** method;@Override  
      **public** View onCreateView(LayoutInflater inflater, ViewGroup parent, Bundle savedInstanceState) {  
       *//* ***Define the xml file for the fragment*****return** inflater.inflate(R.layout.***fragment\_main***, parent, **false**);  
      }
      1. The **onCreateView** method is called when Fragment should create its View object hierarchy either dynamically or via XML layout inflation
      2. Here we are using the **XML layout inflation** on **fragment\_main.xml** we created above
   4. Override the “**onViewCreated**” method;  
      @Override  
      **public void** onViewCreated(View view, Bundle savedInstanceState) {  
       *// Setup any handles to view objects here  
       //adding onclick litner for button to display a toast* Button buttonToast = view.findViewById(R.id.***buttonToast***);  
       buttonToast.setOnClickListener(**new** View.OnClickListener() {  
       @Override  
       **public void** onClick(View v) {  
       Toast.*makeText*(getContext(), **"Toast from Fragment!!!"**, Toast.***LENGTH\_LONG***).show();  
       }  
       });  
      }
      1. This event is triggered soon after **onCreateView()**.
      2. Any view setup should occur here.
         1. E.g., view lookups and attaching view listeners.
      3. Set the **OnClickListener** of the **buttonToast** inside this method
2. Embed the Fragment in an **MainActivity**
   1. There are two ways to add a fragment to an activity:
      1. dynamically using Java
      2. statically using XML.
   2. Any activity using fragments should make sure to extend from **FragmentActivity** or **AppCompatActivity**
   3. Open “**MainActivity**” inside “java” folder, observe it already extends “**AppCompatActivity**”
   4. Use the **FragmentManager** to create a **FragmentTransaction** which allows us to add fragments to the **FrameLayout** at runtime
   5. Inside the “**onCreate**” method of the **MainActivity,** initialize the **MainFragment** and add to the “**FrameLayout**” with id “**fragment\_content**” as shown below;  
      ***// Begin the transaction*FragmentTransaction ft = getSupportFragmentManager().beginTransaction();  
      *// Replace the contents of the container with the new fragment*ft.replace(R.id.*fragment\_content*, new MainFragment());  
      *// or ft.add(R.id.content\_frame, new MainFragment());  
      // Complete the changes added above*ft.commit();**
   6. The complete method will now look like below;  
      
3. Now, save everything, run the app and check. It should appear like below;  
   

# Sensors in Android

Many Android-powered devices include built-in sensors that measure motion, orientation, and environmental conditions such as ambient light or temperature.

The Android sensor framework lets developers to access many types of sensors. Mainly two types of sensors;

* **hardware-based**
  + These are physical components built into a handset or tablet device.
  + They derive their data by directly measuring specific environmental properties, such as acceleration, geomagnetic field strength, or angular change.
* **software-based**.
  + These sensors are not physical devices, although they mimic hardware-based sensors.
  + Software-based sensors derive their data from one or more of the hardware-based sensors and are sometimes called **virtual sensors** or **synthetic sensors**.
  + The **linear** **acceleration sensor** and the **gravity sensor** are examples of software-based sensors.

Some Android-powered devices have every type of sensor.   
E.g.:- most handset devices and tablets have an accelerometer and a magnetometer, but fewer devices have barometers or thermometers.

Also, a device can have more than one sensor of a given type. For example, a device can have two gravity sensors, each one having a different range.

## Sensor types supported by the Android platform.

|  |  |  |  |
| --- | --- | --- | --- |
| Sensor | Type | Description | Common Uses |
| TYPE\_ACCELEROMETER | Hardware | Measures the acceleration force in m/s2 that is applied to a device on all three physical axes (x, y, and z), including the force of gravity. | Motion detection (shake, tilt, etc.). |
| TYPE\_AMBIENT\_TEMPERATURE | Hardware | Measures the ambient room temperature in degrees Celsius (°C). See note below. | Monitoring air temperatures. |
| TYPE\_GRAVITY | Software or Hardware | Measures the force of gravity in m/s2 that is applied to a device on all three physical axes (x, y, z). | Motion detection (shake, tilt, etc.). |
| TYPE\_GYROSCOPE | Hardware | Measures a device's rate of rotation in rad/s around each of the three physical axes (x, y, and z). | Rotation detection (spin, turn, etc.). |
| TYPE\_LIGHT | Hardware | Measures the ambient light level (illumination) in lx. | Controlling screen brightness. |
| TYPE\_LINEAR\_ACCELERATION | Software or Hardware | Measures the acceleration force in m/s2 that is applied to a device on all three physical axes (x, y, and z), excluding the force of gravity. | Monitoring acceleration along a single axis. |
| TYPE\_MAGNETIC\_FIELD | Hardware | Measures the ambient geomagnetic field for all three physical axes (x, y, z) in μT. | Creating a compass. |
| TYPE\_ORIENTATION | Software | Measures degrees of rotation that a device makes around all three physical axes (x, y, z). As of API level 3 you can obtain the inclination matrix and rotation matrix for a device by using the gravity sensor and the geomagnetic field sensor in conjunction with the getRotationMatrix() method. | Determining device position. |
| TYPE\_PRESSURE | Hardware | Measures the ambient air pressure in hPa or mbar. | Monitoring air pressure changes. |
| TYPE\_PROXIMITY | Hardware | Measures the proximity of an object in cm relative to the view screen of a device. This sensor is typically used to determine whether a handset is being held up to a person's ear. | Phone position during a call. |
| TYPE\_RELATIVE\_HUMIDITY | Hardware | Measures the relative ambient humidity in percent (%). | Monitoring dewpoint, absolute, and relative humidity. |
| TYPE\_ROTATION\_VECTOR | Software or Hardware | Measures the orientation of a device by providing the three elements of the device's rotation vector. | Motion detection and rotation detection. |
| TYPE\_TEMPERATURE | Hardware | Measures the temperature of the device in degrees Celsius (°C). This sensor implementation varies across devices and this sensor was replaced with the TYPE\_AMBIENT\_TEMPERATURE sensor in API Level 14 | Monitoring temperatures. |

## Sensor Categories available in Android

The android platform supports three broad categories of sensors.

* Motion Sensors
  + Are used to infer complex user gestures and motions
  + These sensors measure acceleration forces and rotational forces along three axes. This category includes accelerometers, gravity sensors, gyroscopes, and rotational vector sensors.
    - E.g. :- such tilt, shake, or rotation
* Environmental sensors
  + Are used measure various environmental parameters near the device
    - E.g.:- ambient air temperature and pressure, illumination, and humidity.
  + This category includes barometers, photometers, and thermometers.
* Position sensors
  + Are used to measure the physical positioning of the device
  + This category Includes orientation sensors and magnetometers.

These sensors can provide data to apps with high precision and accuracy.

The **device camera**, **fingerprint sensor**, **microphone, and GPS (location) sensor** all have **their own APIs** and **are NOT considered** part of the Android sensor framework.

## Android Sensor Framework

Android sensor framework allows the developer to access these sensors and acquire raw sensor data. The sensor framework is part of the **android.hardware** package and includes the following classes and interfaces;

### SensorManager (**android.hardware.SensorManager**)

* This class is used by **developers** to get access to the sensors available within the Android platform.
* Not every Android-equipped device will support all of the sensors in the SensorManager.
* This class can be used to create an instance of the sensor service.
* Provides various methods for
  + Accessing and listing sensors.
  + Registering and unregistering sensor event listeners.
  + Acquiring orientation information.
* This class also provides several sensor constants that are used to
  + Identify Sensor
    - Sensor type - Orientation, accelerometer, light, magnetic field, proximity, temperature, etc
  + Report sensor accuracy.
    - Accuracy - High, low, medium, unreliable
  + Set data acquisition rates.
    - Sampling rate - Fastest, game, normal, user interface
      * When an application requests a specific sampling rate, it is really only a hint, or suggestion, to the sensor subsystem.
      * There is no guarantee of a particular rate being available.
  + Calibrate sensors.
* An instance of the class can be created as below;  
  ***SensorManager sensorManager = (SensorManager)this.getSystemService(SENSOR\_SERVICE);***

### Sensor (**android.hardware.Sensor)**

* This class is used by **developers** to create an instance of a specific sensor
  + By calling the **getDefaultSensor()** method of SensorManager class.
* It provides various methods to use a sensor's capabilities.
* An instance of the class can be created as below;  
  ***Sensor lightSensor = sensorManager.getDefaultSensor(Sensor.TYPE\_LIGHT);***

### SensorEvent (**android.hardware.SensorEvent)**

* The **system** uses this class to create a sensor event object.
* It provides information about a sensor event.
* A sensor event object includes the following information:
* The raw sensor data.
* The type of sensor that generated the event.
* The accuracy of the data.
* The timestamp for the event.

### SensorEventListener (**android.hardware.SensorEventListener)**

* This interface is used by **developers** to create two callback methods that receive notifications (sensor events).
* When sensor values change. --> **onSensorChanged()**
* When sensor accuracy change --> **onAccuracyChanged()**
* Once a sensor is declared, a listener should be registered for it and override two methods which are onAccuracyChanged and onSensorChanged, as shown below;  
  ***sensorManager.registerListener(this, lightSensor, SensorManager.SENSOR\_DELAY\_NORMAL);  
  public void onAccuracyChanged(Sensor sensor, int accuracy) {  
  }  
  public void onSensorChanged(SensorEvent event) {***

***}***

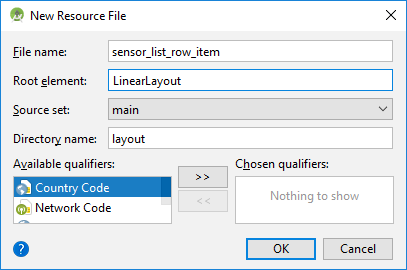
Next lets see how to use sensors in our app;

### List the available sensors of Device

We can get a list of sensors supported by the device in which the app is running, by calling the **getSensorList()** method of the SensorManager, which will return a list of sensors containing their name and version number and much more information.

We can then iterate the list to get the information, for that a new UI is required.

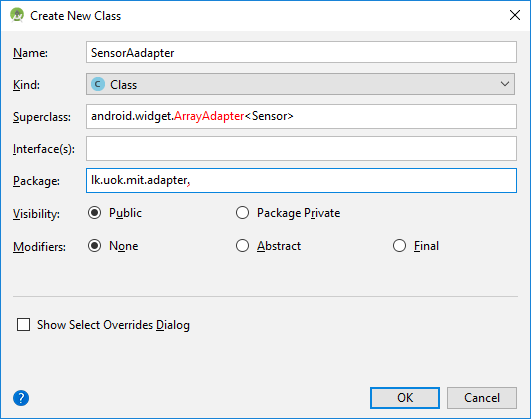
#### Design a single item of the ListView

1. To display the available list of sensors in device, first lets define a layout to represent one row of the ListView
   1. Got to “rest/layout” folder, right click on it and select “New”🡪”Layout resource File”
   2. Fill the details like below;  
      
      1. File Name:- **sensor\_list\_row\_item**
      2. Root Element:- **LinearLayout**
      3. Source Set:-**main**
      4. Directory Name:- **layout**
   3. Open the “sensor\_list\_row\_item.xml” file and do the below modifications;
      1. Change the **orientation** of **LinearLayout** to “**horizontal**”
      2. Add three “TextView” s to display the “Name”, “Vendor” and “Version” of sensors available in the device as shown below;  
         *<?***xml version="1.0" encoding="utf-8"***?>*<**LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
          android:orientation="horizontal"  
          android:layout\_width="match\_parent"  
          android:layout\_height="match\_parent"**>  
           
          <**TextView  
          android:id="@+id/textViewSensorName"  
          android:layout\_width="0px"  
          android:layout\_height="wrap\_content"  
          android:layout\_weight="3"  
          android:textColor="#000000"  
          android:text="Sensor Name"** />  
           
          <**TextView  
          android:id="@+id/textViewSensorVendor"  
          android:layout\_width="0px"  
          android:layout\_height="wrap\_content"  
          android:layout\_weight="2"  
          android:textColor="#000000"  
          android:text="Vendor"** />  
           
          <**TextView  
          android:id="@+id/textViewSensorVersion"  
          android:layout\_width="0px"  
          android:layout\_height="wrap\_content"  
          android:layout\_weight="1"  
          android:textColor="#000000"  
          android:text="Sensor Version"** />  
           
         </**LinearLayout**>

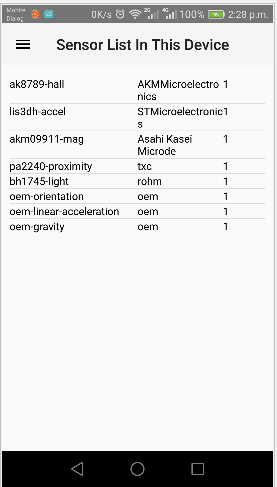
#### Adding a List View to display sensors

1. Add a ListView to the “**fragment\_main.xml**” to display the sensor list
   1. Open “**fragment\_main.xml**” inside “res/layout” and **remove the existing button and text view**
   2. Add a ListView to display the sensors as shown below;  
      *<?***xml version="1.0" encoding="utf-8"***?>*<**LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
       android:orientation="vertical"  
       android:padding="10dp"  
       android:layout\_width="match\_parent"  
       android:layout\_height="wrap\_content"**>  
        
       <**ListView  
       android:id="@+id/listViewSensors"  
       android:layout\_width="match\_parent"  
       android:layout\_height="wrap\_content"  
       android:layout\_alignParentStart="true"  
       android:layout\_alignParentLeft="true"  
       android:layout\_alignParentTop="true"** />  
        
      </**LinearLayout**>

#### Creating a Custom Adapter to render a Single Sensor Row I List

1. Create a custom adapter to display s single row;
   1. Right click on “java” folder and select “New”🡪”Java Class”
   2. Fill in the details as shown below;  
      
      1. Name:-**SensorAdapter**
      2. Kind:-**Class**
      3. Super Class:- **android.widget.ArrayAdapter**
      4. Package:- **lk.uok.mit.adapter**
   3. Open the **“SensorAdapter”** class and add the below code;  
      **package** lk.uok.mit.adapter;  
        
      **import** android.content.Context;  
      **import** android.hardware.Sensor;  
      **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.TextView;  
        
      **import** java.util.List;  
        
      **import** lk.uok.mit.helloworld.R;  
        
      **public class** SensorAadapter **extends** ArrayAdapter<Sensor> {  
        
       ***//a variable to refer to the passed data set through constructor*****private** List<Sensor> **sensors**;  
       ***//a variable to refer to the passed context through constructor*****private** Context **context**;  
        
       ***//four variables to refer to four text boxes*****private** TextView **rowItemSensorName**;  
       **private** TextView **rowItemVendor**;  
       **private** TextView **rowItemSensorVersion**;  
        
       ***//add a constructor to accept context and data set*****public** SensorAadapter(Context context, List<Sensor> sensors) {  
       ***//call super clase's constructor by passing the context, layout and the data set*****super**(context, R.layout.***sensor\_list\_row\_item***, sensors);  
       **this**.**context** = context;  
       **this**.**sensors** = sensors;  
       }  
        
       @Override  
       **public** View getView(**int** position, @Nullable View convertView,  
       @NonNull ViewGroup parent) {  
       ***//Using the “position” parameter get the data item for current  
       // position from sensors list***Sensor currentSensor = **this**.**sensors**.get(position);  
       ***//Check if an existing view is being reused, otherwise inflate the view*****if** (convertView == **null**) {  
       ***//a view to be reused is not available, therefore inflate new views  
       //to inflate views initialize a android.view.LayoutInflater object***LayoutInflater inflater = LayoutInflater.*from*(**this**.**context**);  
       ***//inflate the R.layout.sensor\_list\_row\_item, and assign the return  
       // value to the convertView***convertView = inflater.inflate(R.layout.***sensor\_list\_row\_item***,  
       parent, **false**);  
       }  
        
       ***//using the inflated (initialized) convertView,  
       // get its child text views by id, if they are null*****this**.**rowItemSensorName** = convertView.findViewById(R.id.***textViewSensorName***);  
       **this**.**rowItemVendor** = convertView.findViewById(R.id.***textViewSensorVendor***);  
       **this**.**rowItemSensorVersion** = convertView.findViewById(R.id.***textViewSensorVersion***);  
        
       ***//now set the data to be displayed in each text view*****this**.**rowItemSensorName**.setText(currentSensor.getName());  
       **this**.**rowItemVendor**.setText(currentSensor.getVendor()); **this**.**rowItemSensorVersion**.setText(Integer.*toString*(currentSensor.getVersion()));  
        
       **return** convertView;  
       }  
      }

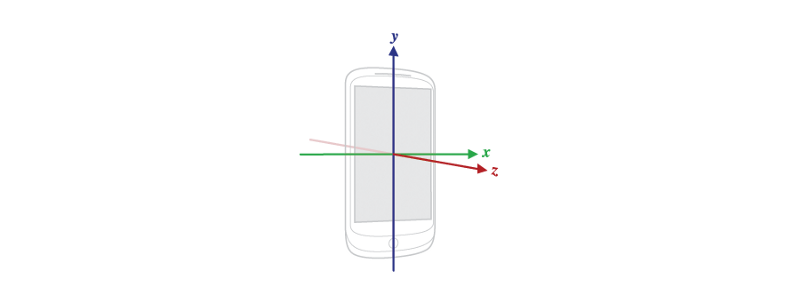
#### Initialize List view, Custom adapter and display the sensor List

1. Call the custom adapter, inflate the sensor list item layout and display sensor list
   1. We will be displaying the sensor list using the “**MainFragment**”, this way, the sensor list will be displayed at the app startup
   2. Open “**MainFragment**” inside “java” folder, and change its code as given below;
   3. Change “**onCreateView**” method, add a line to change the text appears on the title bar of the app (currently its “HelloWorld”)  
        
      *// The onCreateView method is called when Fragment should create its View object hierarchy,  
      // either dynamically or via XML layout inflation.*@Override  
      **public** View **onCreateView**(LayoutInflater inflater, ViewGroup parent, Bundle savedInstanceState) {  
       *//change the title bar text to meaningful one* getActivity().setTitle(**"Sensor List In This Device"**);  
       *// Defines the xml file for the fragment* **return** inflater.inflate(R.layout.***fragment\_main***, parent, **false**);  
      }
   4. Change the “**onViewCreated**” method;
      1. Remove references to the TextView and Button (we removed them from layout)
      2. Get a reference to the ListView
      3. Get the list of sensors in device
      4. initialize the adapter using the retrieved sensor list and context
      5. set the initialized adapter to the list view
      6. call “**notifyDataSetChanged**” method of the adapter to reflect the latest data set available in the “messages”  
           
         @Override  
         **public void** onViewCreated(View view, Bundle savedInstanceState) {  
          *//Get a reference to the ListView* ListView listView = (ListView) view.findViewById(R.id.***listViewSensors***);  
          *//Get the list of sensors in device* SensorManager mSensorManager = (SensorManager) getContext().getSystemService(Context.***SENSOR\_SERVICE***);  
          List<Sensor> sensorList = mSensorManager.getSensorList(Sensor.***TYPE\_ALL***);  
          *//initialize the adapter using the retrieved sensor list and context* SensorAadapter adapter = **new** SensorAadapter(getContext(), sensorList);  
          *//set the initialized adapter to the list view* listView.setAdapter(adapter);  
          *//call “notifyDataSetChanged” method of the adapter to reflect the  
          // latest data set available in the “messages”* adapter.notifyDataSetChanged();  
         }
2. Now Run the app and check the output (list will differ from device to device based on available sensors)  
   

## Using the Accelerometer in Android app

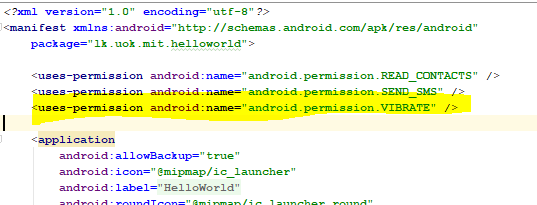
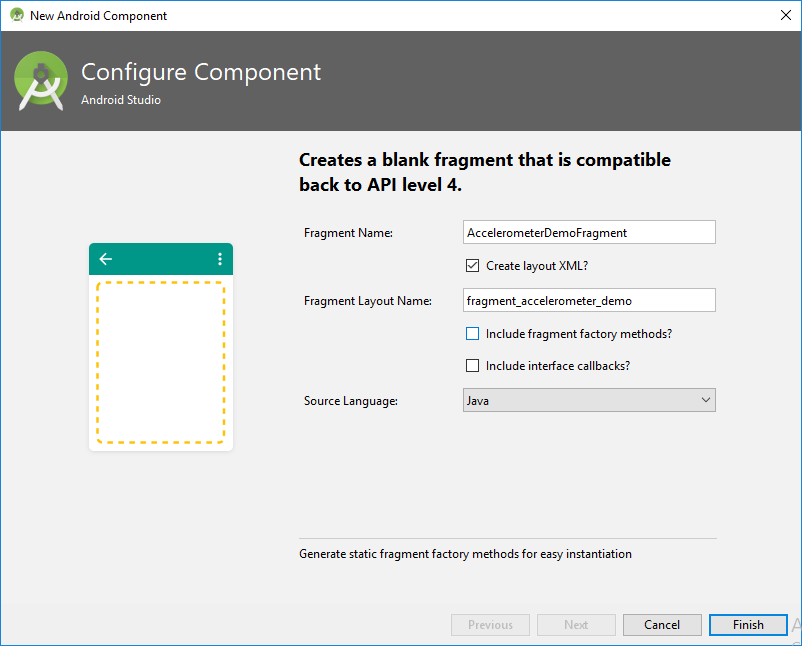
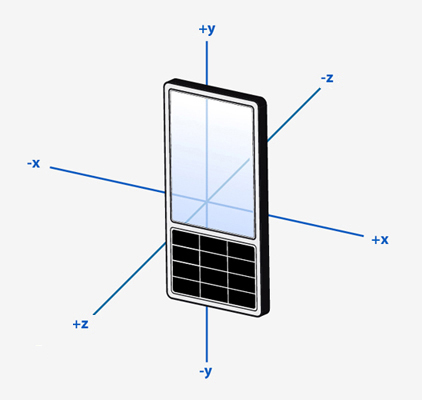
The accelerometer is one of the many hardware sensors of modern smartphones which is available in Android app development.

* With this sensor it’s possible to determinate the orientation of the terminal in the real world.
  + Considering as coordinate axis the middle point of the device.



* The accelerometer calculates the linear acceleration in each of the 3 axes (x, y, z);
* Each axis has its own accelerometer, so that the data can be received individually.

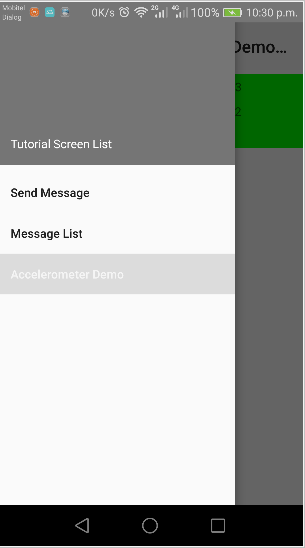
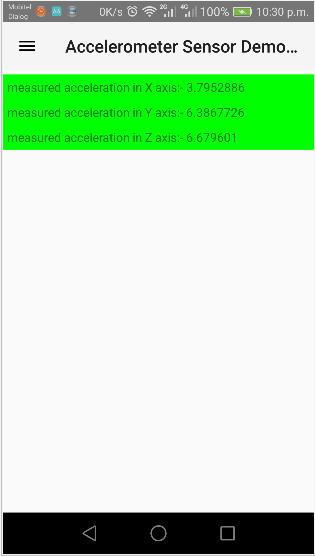
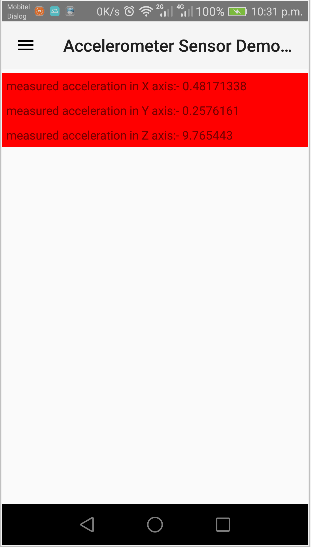
Lets make our app detect changes using accelerometer and vibrates and change background color

1. Add the permission to app to use vibration in our app in “**AndroidManifest.xml**”
   1. Open **AndroidManifest.xml** inside “manifests” folder and add the permission like below;  
      <**uses-permission android:name="android.permission.VIBRATE"** />
   2. Will look like below;  
      
2. Create a new “**Fragment**” to display background color change  
   previously we created a spate layout class and java class for a Fragment and then combined it to get a good understanding of a Fragment.  
   Here let’s use the android Studio’s inbuilt facility to generate a fragment.
   1. Right click on “**java/lk.uok.mit.fragment**”, select “**New**”🡪”**Fragment**”🡪”**Fragment (Blank)**”  
      
   2. Fill the fields as shown below;  
      
      1. Fragment Name:- **AccelerometerDemoFragment**
      2. Create Layout XML:- **Check**
      3. Fragment Layout Name:- **fragment\_accelerometer\_demo**
      4. **Uncheck both “Include fragment factory methods” and “include interface callback”**
      5. Source Language:-Java
3. Add a **menu item** to the **drawer** to access the new “**AccelerometerDemoFragment**”
   1. Open the “**drawer\_view.xml**” inside “res/menu” folder and add a new menu item with id “**nav\_accelerometer\_demo**” like below;  
      *<?***xml version="1.0" encoding="utf-8"***?>*<**menu xmlns:android="http://schemas.android.com/apk/res/android"**>  
       <**group android:checkableBehavior="single"**>  
       <**item  
       android:id="@+id/nav\_send\_message"  
       android:title="Send Message"** />  
       <**item  
       android:id="@+id/nav\_hello\_world"  
       android:title="Message List"** />  
       <**item  
       android:id="@+id/nav\_accelerometer\_demo"  
       android:title="Accelerometer Demo"** />  
       </**group**>  
      </**menu**>
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;  
      *//get a reference to the navigation view*NavigationView navigationView = findViewById(R.id.***nav\_view***);  
      navigationView.setNavigationItemSelectedListener(  
       **new** NavigationView.OnNavigationItemSelectedListener() {  
       @Override  
       **public boolean** onNavigationItemSelected(MenuItem menuItem) {  
       *// set item as selected to persist highlight* menuItem.setChecked(**true**);  
       *// close drawer when item is tapped* **mDrawerLayout**.closeDrawers();  
       *// Add code here to update the UI based on the item selected  
       //to determine which menu item is clicked, add a switch* Intent intent = **null**;  
       **switch** (menuItem.getItemId()) {  
       **case** R.id.***nav\_send\_message***:  
       *//navigate to SendMessageActivity* intent = **new** Intent(**context**, SendMessageActivity.**class**);  
       startActivity(intent);  
       **break**;  
       **case** R.id.***nav\_hello\_world***:  
       *//Navigate to HelloWorldActivity* intent = **new** Intent(**context**, HelloWorldActivity.**class**);  
       startActivity(intent);  
       **break**;  
       **case** R.id.***nav\_accelerometer\_demo***:  
       *//Open the AccelerometerDemoFragment  
       // Begin the transaction* FragmentTransaction ft = getSupportFragmentManager().beginTransaction();  
       *// Replace the contents of the container with the new fragment* ft.replace(R.id.***fragment\_content***, **new** AccelerometerDemoFragment());  
       *// or ft.add(R.id.content\_frame, new MainFragment());  
       // Complete the changes added above* ft.commit();  
       **break**;  
       }  
       **return true**;  
       }  
       });
5. Modify the “**fragment\_accelerometer\_demo.xml**” to display the three values that could be retrieved from the accelerometer,   
   
   1. the device's position in space, the x, y, and z axis.
   2. Open the **fragment\_accelerometer\_demo.xml**  file inside “res/layout” folder and add the code below;  
      <**LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
       xmlns:tools="http://schemas.android.com/tools"  
       android:id="@+id/fragment\_background"  
       android:layout\_width="match\_parent"  
       android:layout\_height="match\_parent"  
       android:orientation="vertical"  
       tools:context="lk.uok.mit.fragment.AccelerometerDemoFragment"**>  
       <**TextView  
       android:id="@+id/textViewX"  
       android:layout\_width="match\_parent"  
       android:layout\_height="wrap\_content"  
       android:layout\_above="@id/textViewY"  
       android:padding="5dp"  
       android:text="Text View X"** />  
        
       <**TextView  
       android:id="@+id/textViewY"  
       android:layout\_width="match\_parent"  
       android:layout\_height="wrap\_content"  
       android:padding="5dp"  
       android:text="Text View Y"** />  
        
       <**TextView  
       android:id="@+id/textViewZ"  
       android:layout\_width="match\_parent"  
       android:layout\_height="wrap\_content"  
       android:layout\_below="@id/textViewY"  
       android:padding="5dp"  
       android:text="Text View Z"** />  
        
      </**LinearLayout**>
6. Modify the “**AccelerometerDemoFragment**” class to listen to the sensor events and get data from the “**Accelerometer Sensor**”
   1. Open the “**AccelerometerDemoFragment.java**” inside “java” folder, and implement “**android.hardware.SensorEventListener**”  
      **public class** AccelerometerDemoFragment **extends** Fragment **implements** SensorEventListener
   2. Implement the two methods to get sensor data as shown below, this is to get rid of compile error we got after implementing **SensorEventListener**, we will add the logic later  
      @Override  
      **public void** onSensorChanged(SensorEvent sensorEvent) {  
        
      }  
      @Override  
      **public void** onAccuracyChanged(Sensor sensor, **int** accuracy) {  
        
      }
   3. Change the title of the fragment inside “**onCreateView**”, and set layout  
      @Override  
      **public** View onCreateView(LayoutInflater inflater, ViewGroup container,  
       Bundle savedInstanceState) {  
       *//set the text appear in title bar* getActivity().setTitle(**"Accelerometer Sensor Demonstration"**);  
       *// Inflate the layout for this fragment* **return** inflater.inflate(R.layout.***fragment\_accelerometer\_demo***, container, **false**);  
      }
   4. Add below class variables to the fragment  
      *//to hold the reference to sensor manager***private** SensorManager **sensorManager**;  
        
      *//to refer to the main LinearLayout layout of the fragment***private** View **mainView**;  
        
      *//to refer to the text view to display measured acceleration in X axis***private** TextView **textViewX**;  
        
      *//to refer to the text view to display measured acceleration in Y axis***private** TextView **textViewY**;  
        
      *//to refer to the text view to display measured acceleration in Z axis***private** TextView **textViewZ**;  
        
      *//to keep if the background color of the parent layout changed, to switch colors***private boolean color** = **false**;  
        
      *//to keep the time in milliseconds on which the UI was updated last time***private long lastUpdate**;  
        
      *//the threshold to determine if the movement of the device is a shake or not***private static final int *SHAKE\_THRESHOLD*** = 600;  
      *//the last position retrieved position of the device along X,Y, and Z axis from accelerometer***private float last\_x**, **last\_y**, **last\_z**;
   5. Add below code to the “**onViewCreated**” method of the **AccelerometerDemoFragment**  
      @Override  
      **public void** onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {  
       **super**.onViewCreated(view, savedInstanceState);  
       *//initialize the parent layout* **mainView** = view.findViewById(R.id.***fragment\_background***);  
       *//set green as the background color at startup* **mainView**.setBackgroundColor(Color.***GREEN***);  
        
       *//initialize the sensor manager* **sensorManager** = (SensorManager) getContext().getSystemService(Context.***SENSOR\_SERVICE***);  
       *//set the current time as the last update time* **lastUpdate** = System.*currentTimeMillis*();  
       *//initialize the three text views* **this**.**textViewX** = view.findViewById(R.id.***textViewX***);  
       **this**.**textViewY** = view.findViewById(R.id.***textViewY***);  
       **this**.**textViewZ** = view.findViewById(R.id.***textViewZ***);  
      }
   6. Override the “**onResume**” method of the fragment and register the current fragment as a listener for sensor events using the sensor manager  
      *//subscribe to listening to sensor events if the fragment execution resumed/started*  
      @Override  
      **public void** onResume() {  
       **super**.onResume();  
       *// register this class as a listener for the orientation and  
       // accelerometer sensors* **sensorManager**.registerListener(**this**,  
       **sensorManager**.getDefaultSensor(Sensor.***TYPE\_ACCELEROMETER***),  
       SensorManager.***SENSOR\_DELAY\_NORMAL***);  
      }
   7. Every time a subscription is done, it should be unsubscribed; override the “**onPause**” method and unsubscribe from listening to sensor events if the fragment execution is paused  
      *//unsubscribe from listening to sensor events if the fragment execution is paused*  
      @Override  
      **public void** onPause() {  
       **super**.onPause();  
       **sensorManager**.unregisterListener(**this**);  
      }
   8. Write a method to vibrate the device
      1. Method name:- **vibrateDevice**
      2. Return Type:-**void**
      3. Parameters:-**none**

Logic:-  
**private void** vibrateDevice() {  
 *//get the vibrator service from system services* Vibrator v = (Vibrator) getContext().getSystemService(Context.***VIBRATOR\_SERVICE***);  
 *// Vibrate for 500 milliseconds* **if** (Build.VERSION.***SDK\_INT*** >= Build.VERSION\_CODES.***O***) {  
 v.vibrate(VibrationEffect.*createOneShot*(500, VibrationEffect.***DEFAULT\_AMPLITUDE***));  
 } **else** {  
 *//deprecated in API 26* v.vibrate(500);  
 }  
}

1. Write a method to get Accelerometer data and update the ui
   1. Method name:- **displayAccelerometerData**
   2. Return Type:- **android.hardware.SensorEvent**
   3. Parameters:-**none**

Logic:-  
**private void** displayAccelerometerData(SensorEvent event) {  
 *//from the android.hardware.SensorEvent, get the x,y and z axis values like below* **float** x = event.**values**[0];  
 **float** y = event.**values**[1];  
 **float** z = event.**values**[2];  
  
 *//get the current time n milliseconds* **long** curTime = System.*currentTimeMillis*();  
  
 *//check if the difference between last update and this update  
 // is greater than 100 ms  
 //this is because the sensors will be continuously producing data,  
 // but we periodically display it* **if** ((curTime - **lastUpdate**) > 100) {  
 *//get the difference between last update and this update* **long** diffTime = (curTime - **lastUpdate**);  
 *//set the last update to current update, as we are upsdaing now* **lastUpdate** = curTime;  
  
 *//set the text of three text boxes in ui to the relieved values from sensor* **this**.**textViewX**.setText(**"measured acceleration in X axis:- "** + x);  
 **this**.**textViewY**.setText(**"measured acceleration in Y axis:- "** + y);  
 **this**.**textViewZ**.setText(**"measured acceleration in Z axis:- "** + z);  
  
 *//use below formula to calculate the speed of movement of device  
 // from last axis data and current axis data* **float** speed = Math.*abs*(x + y + z - **last\_x** - **last\_y** - **last\_z**) / diffTime \* 10000;  
 *//if the speed is greater than the threshold, consider as a shake* **if** (speed > ***SHAKE\_THRESHOLD***) {  
 *//switch the color between red and green for each shake* **if** (**color**) {  
 **mainView**.setBackgroundColor(Color.***GREEN***);  
 } **else** {  
 **mainView**.setBackgroundColor(Color.***RED***);  
 }  
 **color** = !**color**;  
 *//vibrate the device on a shake* vibrateDevice();  
 }  
 *//se the last position to current position* **last\_x** = x;  
 **last\_y** = y;  
 **last\_z** = z;  
 }  
}

1. Call the “**displayAccelerometerData**” method from inside “**onSensorChanged**” method as shown below;  
   @Override  
   **public void** onSensorChanged(SensorEvent event) {  
    *//check if the sensor type is ACCELEROMETER* **if** (event.**sensor**.getType() == Sensor.***TYPE\_ACCELEROMETER***) {  
    *//if yes, write a method to get its data and display* displayAccelerometerData(event);  
    }  
   }
2. Now the coding is completed to get data from acidometer and display in UI and vibrate the device on a shake, save everything, run and check (remember to shake the device)  
     

Source code for this tutorial part can be found in Git Repository given below: - <https://github.com/nadee158/android_tutorial_part_5.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

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

*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