

# Unit-3

Advance GUI Controls

- ▶ Material Design: Using Material Theme, Creating Lists and Cards
- ▶ Navigation Drawer: Create a Drawer Layout, Initialize the Drawer List, Handle Navigation Click Events
- ▶ Working with Drawables: Defining Custom Animations: Customize Touch Feedback, Circular Reveal and Zooming a View: Creating Views, Set up the Zoom Animation
- ▶ Displaying Data using Custom ListView and GridView
- ▶ Creation and Utilization of Charts

# Material Design

- ▶ Material design is a comprehensive guide for visual, motion, and interaction design across platforms and devices.
- ▶ Android now includes support for material design apps.
- ▶ To use material design in your Android apps use the new components and functionality available in Android 5.0 (API level 21) and above.
- ▶ Android provides the following elements for you to build material design apps:
  - ▶ A new theme
  - ▶ New widgets for complex views
  - ▶ New APIs for custom shadows and animations

# Using Material theme

- ▶ The new material theme provides:
  - ▶ System widgets that let you set their color palette
  - ▶ Touch feedback animations for the system widgets
  - ▶ Activity transition animations

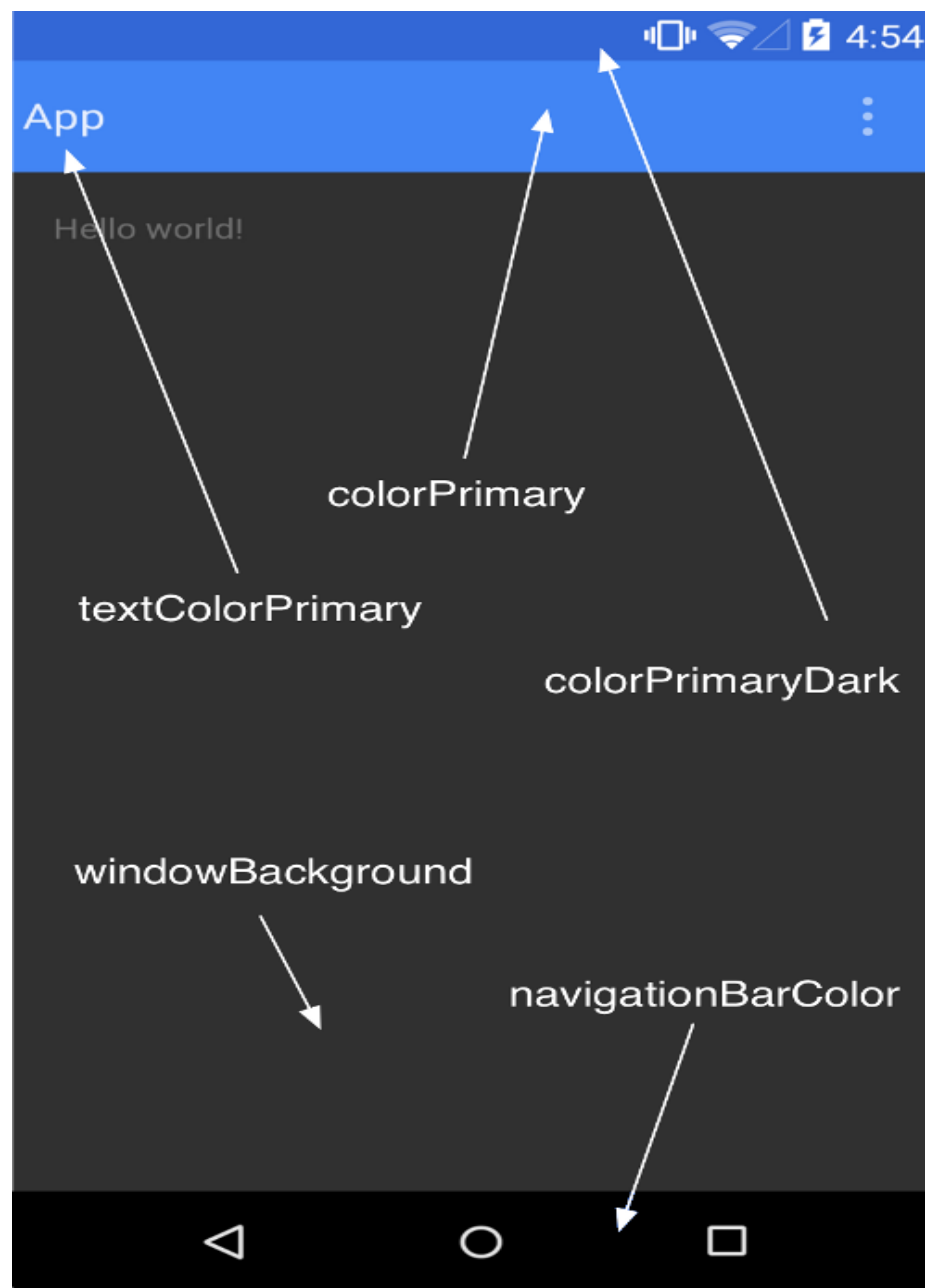
# The material theme is defined as:

- ▶ `@android:style/Theme.Material` (dark version)
- ▶ `@android:style/Theme.Material.Light` (light version)
- ▶ `@android:style/Theme.Material.Light.DarkActionBar`

# Customize the Color Palette

- ▶ To customize the theme's base colors to fit your brand, define your custom colors using theme attributes when you inherit from the material theme:
- ▶ `<resources>`
- ▶ `<!-- inherit from the material theme -->`
- ▶ `<style name="AppTheme" parent="android:Theme.Material">`
- ▶ `<!-- Main theme colors -->`
- ▶ `<!-- your app branding color for the app bar -->`
- ▶ `<item name="android:colorPrimary">@color/primary</item>`
- ▶ `<!-- darker variant for the status bar and contextual app bars -->`
- ▶ `<item name="android:colorPrimaryDark">@color/primary_dark</item>`
- ▶ `<!-- theme UI controls like checkboxes and text fields -->`
- ▶ `<item name="android:colorAccent">@color/accent</item>`
- ▶ `</style>`
- ▶ `</resources>`

**Figure 3.** Customizing the material theme.



# Customize the Status Bar

- ▶ The material theme lets you easily customize the status bar, so you can specify a color that fits your brand and provides enough contrast to show the white status icons.
- ▶ To set a custom color for the status bar, use the `android:statusBarColor` attribute when you extend the material theme.
- ▶ By default, `android:statusBarColor` inherits the value of `android:colorPrimaryDark`.



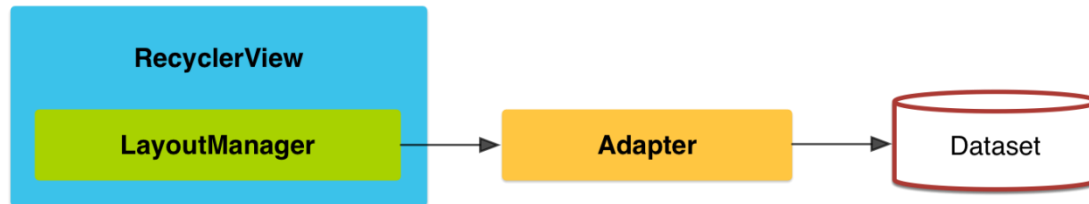
- ▶ You can also draw behind the status bar yourself. For example, if you want to show the status bar transparently over a photo, with a subtle dark gradient to ensure the white status icons are visible.
- ▶ To do so, set the `android:statusBarColor` attribute to `@android:color/transparent` and adjust the window flags as required.
- ▶ You can also use the `Window.setStatusBarColor()` method for animations or fading.

# Creating Lists and Cards

- ▶ To create complex lists and cards with material design styles in your apps, you can use the RecyclerView and CardView widgets.

# Create Lists

- ▶ The RecyclerView widget is a more advanced and flexible version of ListView.
- ▶ This widget is a container for displaying large data sets that can be scrolled very efficiently by maintaining a limited number of views.
- ▶ Use the RecyclerView widget when you have data collections whose elements change at runtime based on user action or network events.
- ▶ The RecyclerView class simplifies the display and handling of large data sets by providing:
  - ▶ Layout managers for positioning items
  - ▶ Default animations for common item operations, such as removal or addition of items
- ▶ You also have the flexibility to define custom layout managers and animations for RecyclerView widgets.



- ▶ A layout manager positions item views inside a RecyclerView and determines when to reuse item views that are no longer visible to the user.
- ▶ To reuse (or recycle) a view, a layout manager may ask the adapter to replace the contents of the view with a different element from the dataset.
- ▶ Recycling views in this manner improves performance by avoiding the creation of unnecessary views or performing expensive `findViewById()` lookups.
- ▶ RecyclerView provides these built-in layout managers:
  - ▶ LinearLayoutManager shows items in a vertical or horizontal scrolling list.
  - ▶ GridLayoutManager shows items in a grid.
  - ▶ StaggeredGridLayoutManager shows items in a staggered grid.
- ▶ To create a custom layout manager, extend the `RecyclerView.LayoutManager` class.

The implementation of RecyclerView requires a few classes to be implemented. The most important classes are listed in the following table

Table 1. Important classes of the RecyclerView API		
Class	Purpose	Optional
Adapter	Provides the data and responsible for creating the views for the individual entry	Required
ViewHolder	Contains references for all views that are filled by the data of the entry	Required
LayoutManager	Contains references for all views that are filled by the data of the entry	Required, but default implementations available
ItemDecoration	Responsible for drawing decorations around or on top of the view container of an entry	Default behavior, but can be overridden
ItemAnimator	Responsible to define the animation if entries are added, removed or reordered	Default behavior, but can be overridden

# Steps:

- ▶ Add the support library to gradle build file.
- ▶ Add the RecyclerView in XML.
- ▶ Specify the Layout Manager type in XML or code.
- ▶ Initialize it inside the activity or fragment
- ▶ Create a layout file for single item inside the RecyclerView.
- ▶ Create an adapter that describes how to display the data for each data item
  - ▶ Data Source[database/XML/JSON/ArrayList]
  - ▶ Number of items
  - ▶ Use the onCreateViewHolder to link the Adapter with the layout .
  - ▶ Use the onBindViewHolder to show data inside the Adapter for each position.
- ▶ Tell the recycler view to use the adapter.

# Detailed Steps of Adapter Class

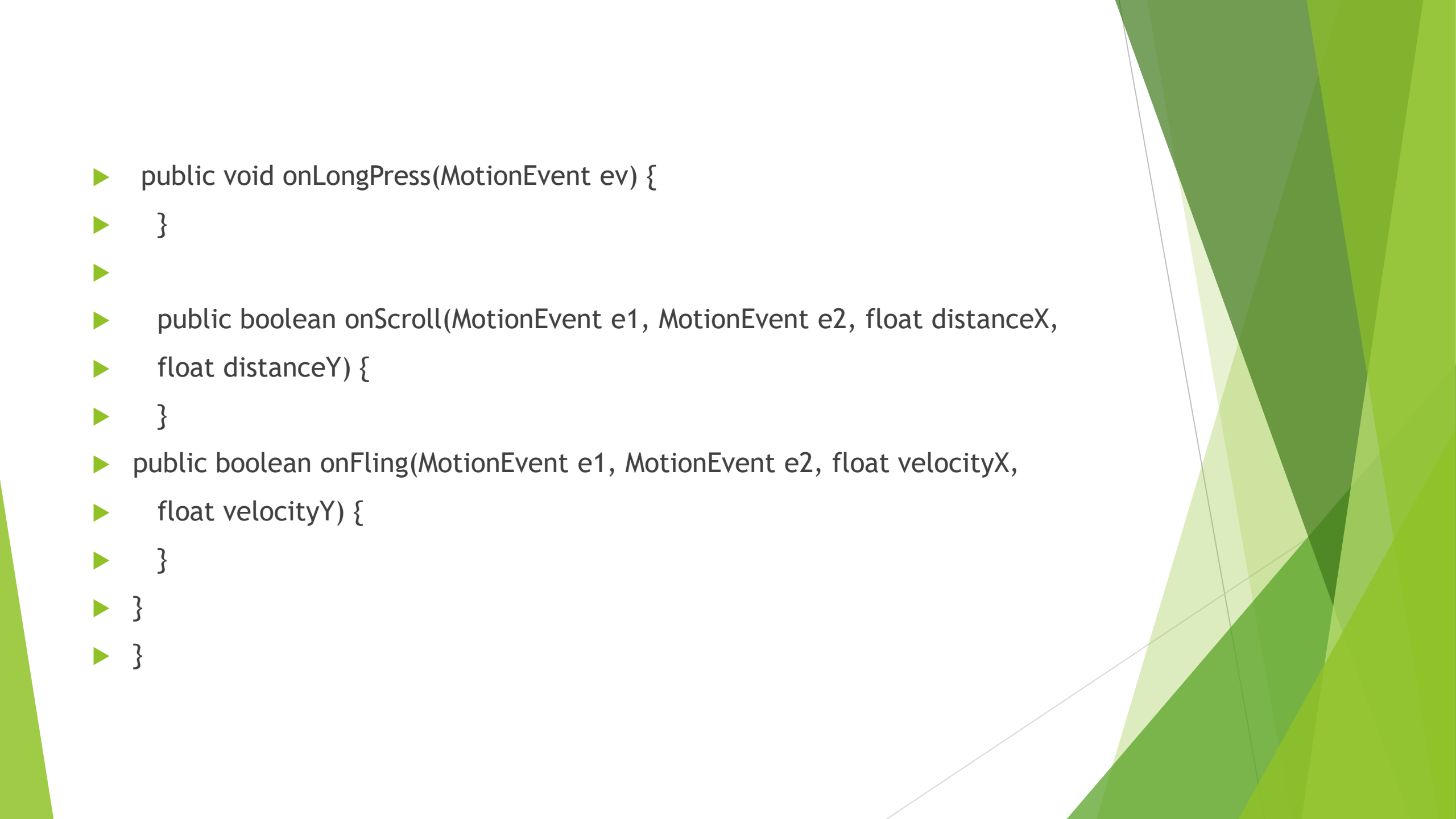
- ▶ The RecyclerView.Adapter class is similar to Adapter class of listview but with ViewHolder class improve performance of RecyclerView.Adapter class.
- ▶ Now, we need to attach textview and image to show with recyclerview. We will create a custom Adapter class and extend Adapter class with RecyclerView.Adapter. This adapter follows the view holder design pattern. Which means we need to define a custom class that extends RecyclerView.ViewHolder.
- ▶ We will create a **getItemCount()** method in the Adapter. This method return the number of items present in the data.
- ▶ Creating a OnCreateViewHolder() Method
- ▶ Next, use the onCreateViewHolder() method. This method is called when the custom ViewHolder needs to be initialized. This method is inflating the layout using LayoutInflater and passing the output to the constructor of the custom ViewHolder.
- ▶

- ▶ Creating a `onBindViewHolder()` Method
- ▶ We will use `onBindViewHolder()` to display the data on a specific position in `RecyclerView`. This method is very similar to `getView` method of `ListView`'s adapter.



# Gesture Detector

- ▶ Android provides special types of touch screen events such as pinch , double tap, scrolls , long presses and flinch. These are all known as gestures.
- ▶ Android provides GestureDetector class to receive motion events and tell us that these events correspond to gestures or not. To use it , you need to create an object of GestureDetector and then extend another class with **GestureDetector.SimpleOnGestureListener** to act as a listener and override some methods. Its syntax is given below –
- ▶ `GestureDetector myG;`
- ▶ `myG = new GestureDetector(this,new Gesture());`
- ▶
- ▶ `class Gesture extends GestureDetector.SimpleOnGestureListener{`
- ▶ `public boolean onSingleTapUp(MotionEvent ev) {`
- ▶ `}`
- ▶

The background of the slide features abstract, overlapping green geometric shapes in various shades, creating a modern and dynamic visual effect.

```
▶ public void onLongPress(MotionEvent ev) {  
▶ }  
▶  
▶ public boolean onScroll(MotionEvent e1, MotionEvent e2, float distanceX,  
▶ float distanceY) {  
▶ }  
▶ public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX,  
▶ float velocityY) {  
▶ }  
▶ }  
▶ }
```

# Create Cards

- ▶ `CardView` extends the `FrameLayout` class and lets you show information inside cards that have a consistent look across the platform. `CardView` widgets can have shadows and rounded corners.
- ▶ To create a card with a shadow, use the `card_view:cardElevation` attribute.
- ▶ Use these properties to customize the appearance of the `CardView` widget:
  - ▶ To set the corner radius in your layouts, use the `card_view:cardCornerRadius` attribute.
  - ▶ To set the corner radius in your code, use the `CardView.setRadius` method.
  - ▶ To set the background color of a card, use the `card_view:cardBackgroundColor` attribute

- ▶ The RecyclerView and CardView widgets are part of the v7 Support Libraries. To use these widgets in your project, add these Gradle dependencies to your app's module:
- ▶ dependencies {
- ▶ ...
- ▶ compile 'com.android.support:cardview-v7:21.0.+'
- ▶ compile 'com.android.support:recyclerview-v7:21.0.+'
- ▶ }

# Navigation Drawer

- ▶ Android navigation drawer is a sliding menu and it's an important UI component. You will see navigation drawer in most of the android applications, it's like navigation menu bars in the websites.
- ▶ Android Navigation Drawer is a sliding left menu that is used to display the important links in the application. Navigation drawer makes it easy to navigate to and fro between those links. It's not visible by default and it needs to be opened either by sliding from left or clicking its icon in the ActionBar.
- ▶ Navigation Drawer is an overlay panel, which is a replacement of an activity screen which was specifically dedicated to show all the options and links in the application.

# Steps to Create Navigation Drawer

- ▶ To implement the Navigation Drawer we first need to add `android.support.v4.widget.DrawerLayout` as the root of the activity layout as shown below.
- ▶ Inside the `DrawerLayout`, add one view that contains the main content for the screen (your primary layout when the drawer is hidden) and another view that contains the contents of the navigation drawer.
- ▶ The menu options in the navigation drawer are stored in the form of a `ListView`. Each option opens in the `FrameLayout`.
- ▶ We've used a **ToolBar** in place of an [ActionBar](#) here. `ToolBar` has been introduced since Android 5.0 as a generalisation of `ActionBar`. It gives us more control and flexibility to modify and its easier to interleave with other views in the hierarchy
- ▶ The layout `ToolBar` is defined in the xml layout
- ▶ We need to use the Theme `Theme.AppCompat.NoActionBar` in the `styles.xml` when using `Toolbars`.
- ▶ The drawer items are stored in the form of a `ListView`. Hence we need to use an `Adapter Class` to provide that data to the activity class.

# Handle Navigation Click Events

- ▶ When the user selects an item in the drawer's list, the system calls `onItemClick()` on the `OnItemClickListener` given to `setOnItemClickListener()`.
- ▶ What you do in the `onItemClick()` method depends on how you've implemented your app structure.

- ▶ However, rather than implementing the `DrawerLayout.DrawerListener`, if your activity includes the action bar, you can instead extend the `ActionBarDrawerToggle` class.
- ▶ The `ActionBarDrawerToggle` implements `DrawerLayout.DrawerListener` so you can still override those callbacks, but it also facilitates the proper interaction behavior between the action bar icon and the navigation drawer.



# ActionBarDrawerToggle

- ▶ This class provides a handy way to tie together the functionality of `DrawerLayout` and the framework `ActionBar` to implement the recommended design for navigation drawers.
- ▶ `ActionBarDrawerToggle` can be used directly as a `DrawerLayout.DrawerListener`, or if you are already providing your own listener, call through to each of the listener methods from your own.

# Defining Custom Animations:

- ▶ Animations in material design give users feedback on their actions and provide visual continuity as users interact with your app. The material theme provides some default animations for buttons and activity transitions, and Android 5.0 (API level 21) and above lets you customize these animations and create new ones:
- ▶ Touch feedback
- ▶ Circular Reveal

# Customize Touch Feedback

- ▶ Touch feedback in material design provides an instantaneous visual confirmation at the point of contact when users interact with UI elements
- ▶ The default touch feedback animations for buttons use the new `RippleDrawable` class, which transitions between different states with a ripple effect.
- ▶ In most cases, you should apply this functionality in your view XML by specifying the view background as:
- ▶ `?android:attr/selectableItemBackground` for a bounded ripple.
- ▶ `?android:attr/selectableItemBackgroundBorderless` for a ripple that extends beyond the view. It will be drawn upon, and bounded by, the nearest parent of the view with a non-null background.
- ▶ Note: `selectableItemBackgroundBorderless` is a new attribute introduced in API level 21.

# Try this...

```
<ripple xmlns:android="http://schemas.android.com/apk/res/android"
  android:color="#ff00ff00">
  <item android:drawable="@android:color/white"/>
</ripple>
```

```
<ripple xmlns:android="http://schemas.android.com/apk/res/android"
  android:color="#ffff0000">
  <item android:id="@android:id/mask"
    android:drawable="@android:color/white" />
</ripple>
```

```
<!-- An unbounded red ripple. -->  
<ripple android:color="#ffff0000" />
```

```
<ripple xmlns:android="http://schemas.android.com/apk/res/android"  
  android:color="#ffff0000">  
  <item android:id="@android:id/mask">  
    <shape android:shape="oval">  
      <solid android:color="#fff" />  
    </shape>  
  </item>  
</ripple>
```

# Circular Reveal

- ▶ Reveal animations provide users visual continuity when you show or hide a group of UI elements. The `ViewAnimationUtils.createCircularReveal()` method enables you to animate a clipping circle to reveal or hide a view.
- ▶ `ViewAnimationUtils` class is used to define common utilities for working with View's animations.

## Method

`createCircularReveal(View view, int centerX, int centerY, float startRadius, float endRadius)`: Returns an `Animator` which can animate a clipping circle

Note that the animation returned here is a one-shot animation. It cannot be re-used, and once started it cannot be paused or resumed. It is also an asynchronous animation that automatically runs off of the UI thread. As a result `onAnimationEnd(Animator)` will occur after the animation has ended.

► Animator.AnimatorListener:

- An animation listener receives notifications from an animation. Notifications indicate animation related events, such as the end or the repetition of the animation.

## Method

onAnimationEnd(Animator animation)  
Notifies the end of the animation.

# Circular Reveal

- ▶ Reveal animations provide users visual continuity when you show or hide a group of UI elements. The `ViewAnimationUtils.createCircularReveal()` method enables you to animate a clipping circle to reveal or hide a view.
- ▶ Steps to give a circular reveal effect.
- ▶ Create a tap on which you need to have circular reveal.
- ▶ Create a View/layout to display and hide for circular reveal.
- ▶ Get the radius of the circular reveal and X, Y coordinates to start the animation from.
- ▶ use `ViewAnimationUtils.createCircularReveal (View view, int centerX, int centerY, float startRadius, float endRadius)`. This method enables us to animate a clipping circle to reveal or hide a view.
  - ▶ It takes following parameters:
  - ▶ `view` The View will be clipped to the animating circle.
  - ▶ `centerX` The x coordinate of the center of the animating circle, relative to view.
  - ▶ `centerY` The y coordinate of the center of the animating circle, relative to view.
  - ▶ `startRadius` The starting radius of the animating circle.
  - ▶ `endRadius` The ending radius of the animating circle



```
// previously invisible view
View myView = findViewById(R.id.my_view);

// get the center for the clipping circle
int cx = myView.getWidth() / 2;
int cy = myView.getHeight() / 2;

// get the final radius for the clipping circle
float finalRadius = (float) Math.hypot(cx, cy);

// create the animator for this view (the start radius is zero)
Animator anim =
    ViewAnimationUtils.createCircularReveal(myView, cx, cy, 0, finalRadius);

// make the view visible and start the animation
myView.setVisibility(View.VISIBLE);
anim.start();
```

# Zooming a View: Creating Views, Set up the Zoom Animation

- ▶ To animate a view from a thumbnail to a full-size image that fills the screen.
- ▶ Creating Views:
- ▶ Create a layout file that contains the small and large version of the content that you want to zoom.
- ▶ **Set up the Zoom Animation :**
- ▶ Once you apply your layout, set up the event handlers that trigger the zoom animation

# Zoom the View :

- ▶ You'll now need to animate from the normal sized view to the zoomed view when appropriate. In general, you need to animate from the bounds of the normal-sized view to the bounds of the larger-sized view. How to implement a zoom animation that zooms from an image thumbnail to an enlarged view by doing the following things:
- ▶ Assign the high-res image to the hidden "zoomed-in" (enlarged) ImageView. The following example loads a large image resource on the UI thread for simplicity. You will want to do this loading in a separate thread to prevent blocking on the UI thread and then set the bitmap on the UI thread. Ideally, the bitmap should not be larger than the screen size.
- ▶ Calculate the starting and ending bounds for the ImageView.
- ▶ Animate each of the four positioning and sizing properties X, Y, (SCALE\_X, and SCALE\_Y) simultaneously, from the starting bounds to the ending bounds. These four animations are added to an AnimatorSet so that they can be started at the same time.
- ▶ Zoom back out by running a similar animation but in reverse when the user touches the screen when the image is zoomed in. You can do this by adding a View.OnClickListener to the ImageView. When clicked, the ImageView minimizes back down to the size of the image thumbnail and sets its visibility to GONE to hide it.

# Custom ListView and GridView

- ▶ In activity\_main.xml include listview /Gridview element.
- ▶ create a layout for the list item that is to be displayed in ListView/GridView.
- ▶ Create custom list class for custom listview/GridView.
- ▶ In main activity define an array that can be displayed as text.
- ▶ Use adapter for data binding.

# Charts (Line, Bar and Pie)

- ▶ There are many libraries available for making charts here we will use MPAndroidChart library.

- ▶ Steps:

1. Add MPAndroidChart library to gradle file .

```
repositories { maven { url "https://jitpack.io" } }  
  
dependencies { compile 'com.github.PhilJay:MPAndroidChart:v2.0.9' }
```

2. Creating a dataset

- All data should be converted into a DataSet object before it can be used by a chart.
- Different types of charts use different subclasses of the DataSet class.
- For example, a BarChart uses a BarDataSet instance.
- Similarly, a PieChart uses a PieDataSet instance.

# Cont..

## ► 2.1 To create a data set you need to have real data .

- Every individual value of the raw data should be represented as an **Entry**.
  - An **ArrayList** of such **Entry** objects is used to create a **DataSet**.
  - So in this case create **BarEntry** objects and add them to an **ArrayList**:
- Now that the **ArrayList** of **Entry** objects is ready, we can create a **DataSet** out of it.
  - `BarDataSet dataset = new BarDataSet(entries, "ANY String");`

3. Create X-axis for the chart. Each x-axis label is represented using a **String** and an **ArrayList** is used to store all the labels.

4. Create a chart:

All charts of this library are subclasses of **ViewGroup**, which means that you can easily add them to any layout. You can define your chart using an XML file or Java code.

```
<com.github.mikephil.charting.charts.BarChart  
    android:id="@+id/barchart1"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"/>
```

# Cont..

## 4. Create a chart:

All charts of this library are subclasses of `ViewGroup`, which means that you can easily add them to any layout. You can define your chart using an XML file or Java code.

```
<com.github.mikephil.charting.charts.BarChart  
    android:id="@+id/barchart1"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"/>
```

```
BarChart barchart1 = (BarChart) findViewById(R.id.barchart1);
```

## 5. Add data to the chart:

```
BarData dt=new BarData(labels,dataset);
```

Set the data:  
`barchart1.setData(dt);`

# Cont..

## ► 6. Add Description:

```
barchart1.setDescription("First BARChart");
```

## ► 7. Add Colors to charts:

- If you do not like the default colors, you can use the **DataSet** class's **setColors** method to change the color scheme.
- However, **MPAndroidChart** also comes with a number of predefined color templates that let you change the look and feel of your data set without having to deal with the individual color values.

- `dataset.setColors(ColorTemplate.COLORFUL_COLORS);`

- Some templates of this library are:

`ColorTemplate.LIBERTY_COLORS`

`ColorTemplate.COLORFUL_COLORS`

`ColorTemplate.JOYFUL_COLORS`

`ColorTemplate.PASTEL_COLORS`

`ColorTemplate.VORDIPLOM_COLORS`



# Cont..

## ► 8. Add Animations to your chart

- All charts of this library support animations, which you can use to make your charts appear more lively.
- The `animateXY( )` method is used to animate both axes of the chart.
- If you want to animate only one of the axes, you can use `animate( )` or `animate( )` to animate the x-axis or y-axis respectively.
- You have to specify the duration (in milliseconds) of the animation when you call these methods