

CHAPTER – 4

ANDROID USER INTERFACE

INTRODUCTION TO MULTIPLE SCREEN SIZE AND ORIENTATION INTERFACES:

Android runs on a variety of devices that offer different screen sizes and densities. For applications, the Android system provides a consistent development environment across devices and handles most of the work to adjust each application's user interface to the screen on which it is displayed.

At the same time, the system provides APIs that allow us to control our application's UI for specific screen sizes and densities, in order to optimize our UI design for different screen configurations. For example, we might want a UI for tablets that's different from the UI for handsets.

Although the system performs scaling and resizing to make our application work on different screens, we should make the effort to optimize our application for different screen sizes and densities. In doing so, we maximize the user experience for all devices and our users believe that our application was actually designed for *their* devices rather than simply stretched to fit the screen on their devices.

For supporting multiple screen sizes and orientation following terms has to be considered:

Screen Size:

- ❖ Actual physical size, measured as the screen's diagonal.
- ❖ For simplicity, Android groups all actual screen sizes into four generalized sizes: small, normal, large, and extra-large.

Screen Density:

- ❖ The quantity of pixels within a physical area of the screen; usually referred to as dpi (dots per inch). For example, a "low" density screen has fewer pixels within a given physical area, compared to a "normal" or "high" density screen.
- ❖ For simplicity, Android groups all actual screen densities into four generalized densities: low, medium, high, and extra high.

Orientation:

- ❖ The orientation of the screen from the user's point of view.
- ❖ This is either landscape or portrait, meaning that the screen's aspect ratio is either wide or tall, respectively. Be aware that not only do different devices operate in different orientations by default, but the orientation can change at runtime when the user rotates the device.

Resolution:

- ❖ The total number of physical pixels on a screen.
- ❖ When adding support for multiple screens, applications do not work directly with resolution; applications should be concerned only with screen size and density, as specified by the generalized size and density groups.

Density-Independent Pixel (dp):

- ❖ A virtual pixel unit that we should use when defining UI layout, to express layout dimensions or position in a density-independent way.
- ❖ The density-independent pixel is equivalent to one physical pixel on a 160 dpi screen, which is the baseline density assumed by the system for a "medium" density screen.
- ❖ At runtime, the system transparently handles any scaling of the dp units, as necessary, based on the actual density of the screen in use.
- ❖ The conversion of dp units to screen pixels is simple: $px = dp * (dpi / 160)$. For example, on a 240 dpi screen, 1 dp equals 1.5 physical pixels. You should always use dp units when defining your application's UI, to ensure proper display of your UI on screens with different densities.

Range of Screens Supported:

Starting with Android 1.6 (API Level 4), Android provides support for multiple screen sizes and densities, reflecting the many different screen configurations that a device may have. We can use features of the Android system to optimize our application's user interface for each screen configuration and ensure that our application not only renders properly, but provides the best user experience possible on each screen.

To simplify the way that we design our user interfaces for multiple screens, Android divides the range of actual screen sizes and densities into:

- ❖ A set of four generalized **sizes**: *small*, *normal*, *large*, and *xlarge*
- ❖ A set of six generalized **densities**:
 - *ldpi* (low) ~120dpi
 - *mdpi* (medium) ~160dpi
 - *hdpi* (high) ~240dpi
 - *xhdpi* (extra-high) ~320dpi
 - *xxhdpi* (extra-extra-high) ~480dpi
 - *xxxhdpi* (extra-extra-extra-high) ~640dpi

The generalized sizes and densities are arranged around a baseline configuration that is a *normal* size and *mdpi* (medium) density. This baseline is based upon the screen configuration for the first Android-powered device, the T-Mobile G1, which has an HVGA screen (until Android 1.6, this was the only screen configuration that Android supported).

Each generalized size and density spans a range of actual screen sizes and densities. For example, two devices that both report a screen size of *normal* might have actual screen sizes and aspect ratios that are slightly different when measured by hand.

Similarly, two devices that report a screen density of *hdpi* might have real pixel densities that are slightly different. Android makes these differences abstract to applications, so we can provide UI designed for the generalized sizes and densities and let the system handle any final adjustments as necessary.

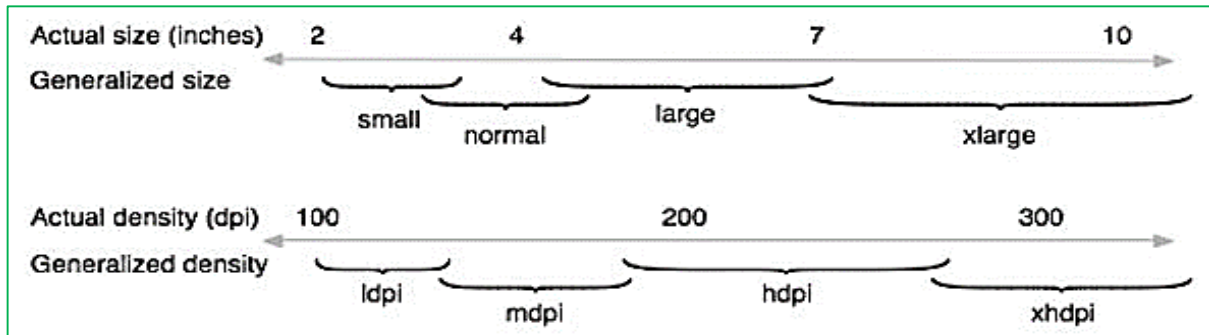


Fig: Illustration of How Android Roughly Maps Actual Sizes and Densities to Generalized Sizes and Densities

As we design our UI for different screen sizes, we'll discover that each design requires a minimum amount of space. So, each generalized screen size above has an associated minimum resolution that's defined by the system. These minimum sizes are in "dp" units the same units we should use when defining our layouts which allows the system to avoid worrying about changes in screen density.

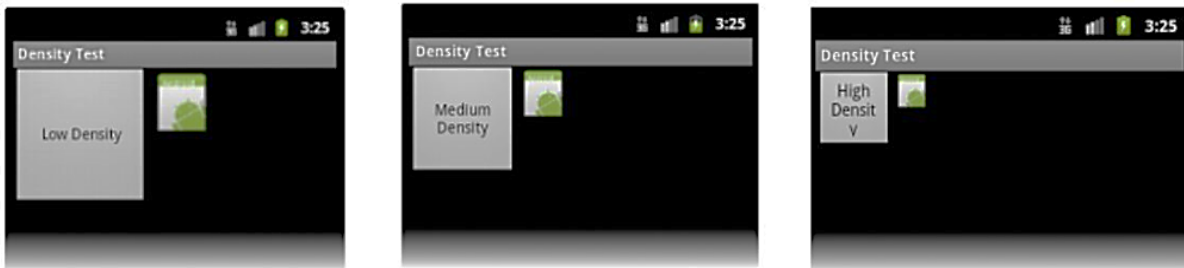
- ❖ *xlarge* screens are at least 960dp x 720dp
- ❖ *large* screens are at least 640dp x 480dp
- ❖ *normal* screens are at least 470dp x 320dp
- ❖ *small* screens are at least 426dp x 320dp

To optimize our application's UI for the different screen sizes and densities, we can provide alternative resources for any of the generalized sizes and densities. Typically, we should provide alternative layouts for some of the different screen sizes and alternative bitmap images for different screen densities. At runtime, the system uses the appropriate resources for our application, based on the generalized size or density of the current device screen.

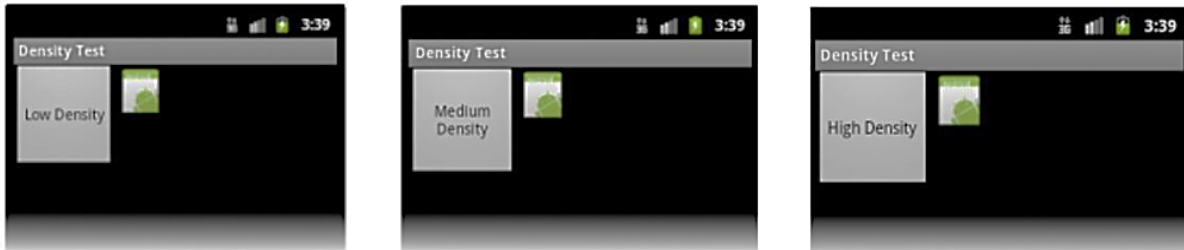
We do not need to provide alternative resources for every combination of screen size and density. The system provides robust compatibility features that can handle most of the work of rendering our application on any device screen, provided that we've implemented our UI using techniques that allow it to gracefully resize.

Density Independence:

Our application achieves "density independence" when it preserves the physical size (from the user's point of view) of user interface elements when displayed on screens with different densities. Maintaining density independence is important because, without it, a UI element (such as a button) appears physically larger on a low-density screen and smaller on a high-density screen. Such density related size changes can cause problems in our application layout and usability.



Example Application Without Support For Different Densities, As Shown On Low, Medium, And High-density Screens.



Example Application With Good Support For Different Densities (It's Density Independent), As Shown On Low, Medium, And High Density Screens.

The Android system helps our application achieve density independence in two ways:

- ❖ The system scales dp units as appropriate for the current screen density.
- ❖ The system scales drawable resources to the appropriate size, based on the current screen density, if necessary

How to Support Multiple Screens:

The foundation of Android's support for multiple screens is its ability to manage the rendering of an application's layout and bitmap drawables in an appropriate way for the current screen configuration. The system handles most of the work to render our application properly on each screen configuration by scaling layouts to fit the screen size/density and scaling bitmap drawables for the screen density, as appropriate. To more gracefully handle different screen configurations, however, we should also:

1. Explicitly Declare in The Manifest Which Screen Sizes Our Application Supports:

By declaring which screen sizes our application supports, we can ensure that only devices with the screens we support can download our application. Declaring support for different screen sizes can also affect how the system draws our application on larger screens specifically, whether our application runs in screen compatibility mode. To declare the screen sizes our application supports, we should include the <supports-screens> element in our manifest file.

2. Provide Different Layouts for Different Screen Sizes:

By default, Android resizes our application layout to fit the current device screen. In most cases, this works fine. In other cases, our UI might not look as good and might need adjustments for different screen sizes. For example, on a larger screen, we might want to adjust the position and size of some elements to take advantage of the additional screen space, or on a smaller screen, we might need to adjust sizes so that everything can fit on the screen. The configuration qualifiers we can use to provide size-specific resources are

small, normal, large and xlarge. For example, layouts for an extra-large screen should go in layout-xlarge.

3. Provide Different Bitmap Drawables for Different Screen Densities:

By default, Android scales our bitmap drawables (.png, .jpg, and .gif files) and Nine-Patch drawables (.9.png files) so that they render at the appropriate physical size on each device. For example, if our application provides bitmap drawables only for the baseline, medium screen density (mdpi), then the system scales them up when on a high-density screen, and scales them down when on a low-density screen. This scaling can cause artifacts in the bitmaps. To ensure our bitmaps look their best, we should include alternative versions at different resolutions for different screen densities.

USER INTERFACE CLASSES:

All user interface elements in an Android app are built using View and ViewGroup objects.

A View is a widget that has an appearance on screen that the user can interact with. Examples of widgets are buttons, labels, text boxes, etc. A View derives from the base class **android.view.View**

A ViewGroup (which is by itself is a special type of View) provides the layout in which we can order the appearance and sequence of views. It provides invisible container that hold other Views or other ViewGroups and define their layout properties. Examples of Viewgroups are LinearLayout, FrameLayout, etc. A ViewGroup derives from the base class **android.view.ViewGroup**.

Attaching Layouts to Java Code:

Assume *res/layout/main.xml* has been created. This layout could be called by an application using the statement **setContentView(R.layout.main)**.

Individual widgets, such as *myButton* could be accessed by the application using the statement *findViewById(...)* as in **Button btn= (Button) findViewById(R.id.myButton);** Where **R** is a class automatically generated to keep track of resources available to the application. In particular R.id... is the collection of widgets defined in the XML layout.

Attaching Listeners to the Widgets:

The button of our example could now be used, for instance a listener for the click event could be written as:

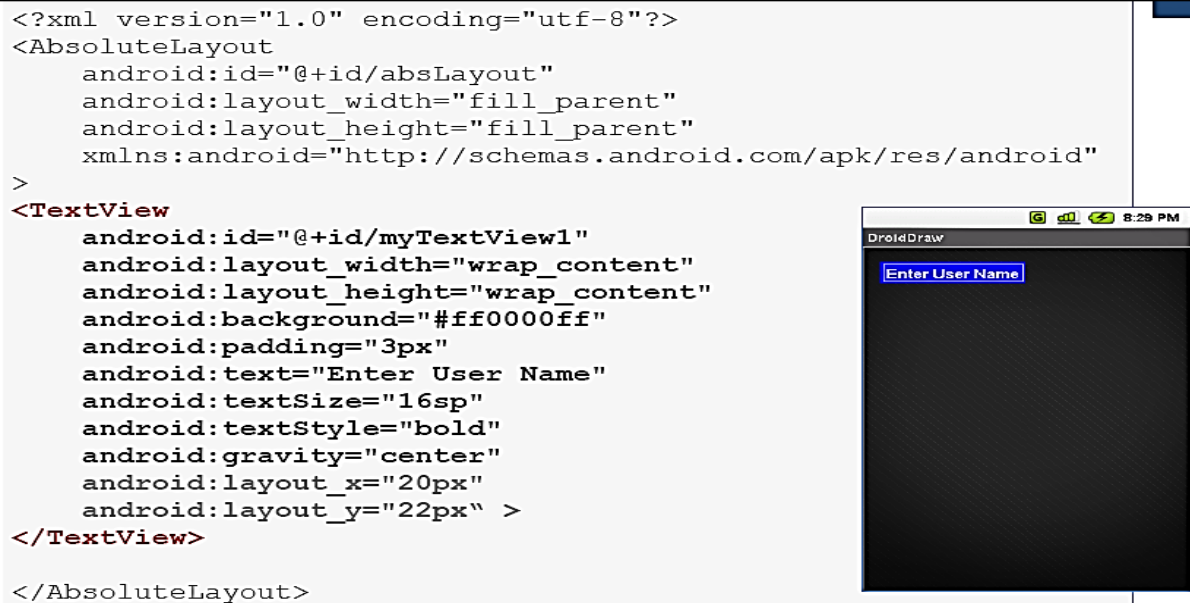
```
btn.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        updateTime();
    }
});

private void updateTime() {
    btn.setText(new Date().toString());
}
```

Basic Widgets:

1. Labels:

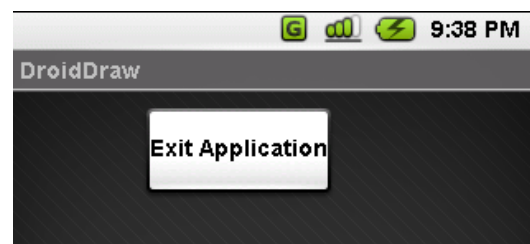
A label is called in android a **TextView**. TextViews are typically used to display a caption. TextViews are *not* editable, therefore they take no input.



2. Button:

A **Button** widget allows the simulation of a clicking action on a GUI. Button is a subclass of TextView. Therefore, formatting a Button's face is similar to the setting of a TextView.

```
<Button
    android:id="@+id/btnExitApp"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="10px"
    android:layout_marginLeft="5px"
    android:text="Exit Application"
    android:textSize="16sp"
    android:textStyle="bold"
    android:gravity="center"
    android:layout_gravity="center_horizontal">
</Button>
```

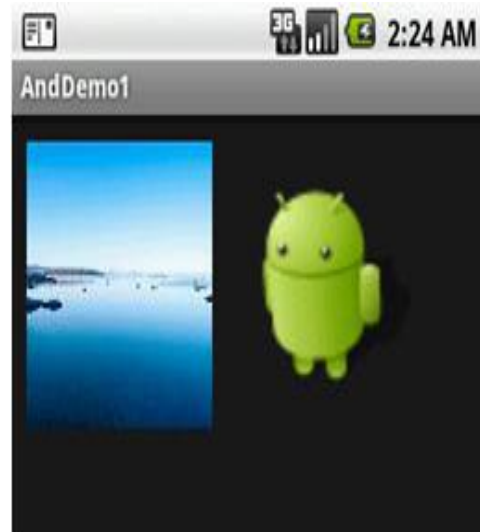


3. Image:

ImageView and **ImageButton** are two Android widgets that allow embedding of images in our applications. Each widget takes an **android:src** or **android:background** attribute (in an XML layout) to specify what picture to use. Pictures are usually reference a drawable resource. We can also set the image content based on a URI from a content

provider via `setImageURI()`. **ImageButton**, is a subclass of `ImageView`. It adds the standard `Button` behavior for responding to click-events.

```
...  
<ImageButton  
    android:id="@+id/myImageBtn1"  
    android:background="@drawable/default_wallpaper"  
    android:layout_width="125px"  
    android:layout_height="131px"  
>  
</ImageButton>  
  
<ImageView  
    android:id="@+id/myImageView1"  
    android:background="@drawable/ic_launcher_android"  
    android:layout_width="108px"  
    android:layout_height="90px"  
>  
</ImageView>
```

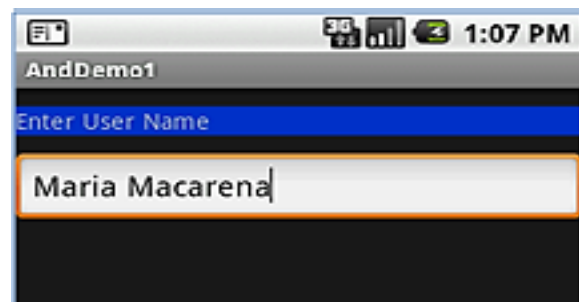


4. EditText:

The **EditText** (or `textBox`) widget is an extension of `TextView` that allows updates. The control configures itself to be *editable*. Important Java methods are:

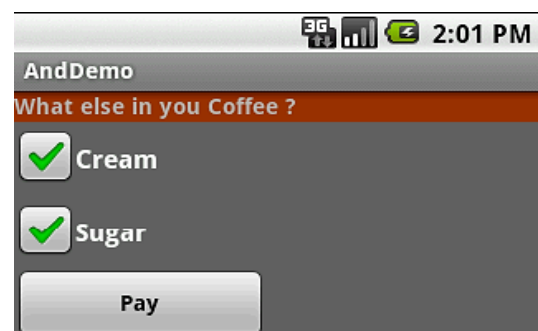
- ❖ `textBox.setText("someValue")`
- ❖ `textBox.getText().toString()`

```
<EditText  
    android:id="@+id/txtUserName"  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:textSize="18sp"  
    android:autoText="true"  
    android:capitalize="words"  
    android:hint="First Last Name"  
</EditText>
```



5. Checkbox:

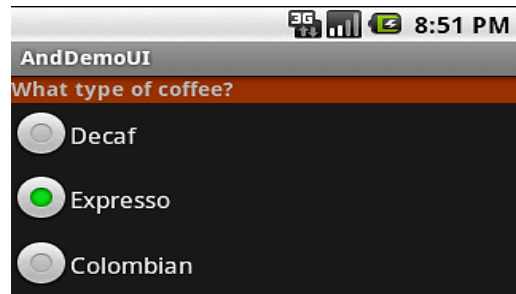
A checkbox is a specific type of two-states button that can be either *checked* or *unchecked*. An example usage of a checkbox inside your activity would be the following:



6. RadioButton:

A radio button is a two-states button that can be either **checked** or **unchecked**. When the radio button is unchecked, the user can press or click it to check it. Radio buttons are normally used together in a **RadioGroup**.

When several radio buttons live inside a radio group, checking one radio button **unchecks** all the others. Similarly, you can call **isChecked()** on a **RadioButton** to see if it is selected, **toggle()** to select it, and so on, like you can with a **CheckBox**.



Unit Measurement in Android:

- ❖ px = Pixels – corresponds to actual pixels on the screen.
- ❖ pt = Points – 1/72 of an inch based on the physical size of the screen.

Density-independent Pixels (dp):

An abstract unit that is based on the physical density of the screen. These units are relative to a 160 dpi screen, so one dp is one pixel on a 160 dpi screen. The ratio of dp-to-pixel will change with the screen density, but not necessarily in direct proportion. Note: The compiler accepts both “dip” and “dp”, though “dp” is more consistent with “sp”.

Scale-independent Pixels (sp):

This is like the dp unit, but it is also scaled by the user’s font size preference. It is recommending us to use this unit when specifying font sizes, so they will be adjusted for both the screen density and user’s preference.

ANDROID XML LAYOUTS, RESOURCES AND STYLES:

Android XML:

XML stands for Extensible Mark-up Language. XML is a very popular format and commonly used for sharing data on the internet. Android provides three types of XML parsers which are **DOM**, **SAX** and **XMLPullParser**. Among all of them android recommend XMLPullParser because it is efficient and easy to use. So we are going to use XMLPullParser for parsing XML.

The first step is to identify the fields in the XML data in which we are interested in. For example, In the XML given below we interested in getting temperature only.

```
<?xml version="1.0"?>
<current>
  <city id="2643743" name="London">
    <coord lon="-0.12574" lat="51.50853"/>
    <country>GB</country>
    <sun rise="2013-10-08T06:13:56" set="2013-10-08T17:21:45"/>
  </city>
```



```

    <temperature value="289.54" min="289.15" max="290.15" unit="kelvin"/>
    <humidity value="77" unit="%"/>
    <pressure value="1025" unit="hPa"/>
</current>

```

Android layout:

A Layout dictates the alignment of widgets (such as Text, Buttons, EditText box) as we see in the Android Application. All the visual structure we see in an android app is designed in a Layout. Every Layout is defined in an xml file which is located in **App > res > Layout** in New Android Studio. Controls location of Views in that ViewGroup.

Types of Layouts:

1. Linear Layout:

Linear Layout is a layout which aligns the widgets or elements in a linear (Straight) fashion. Linear Layout consists of two types of orientation:

- ❖ Vertical Orientation
- ❖ Horizontal Orientation

Example:

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />
</LinearLayout>

```

2. Relative Layout:

Relative Layout is a layout where the widgets (such as Text Views, Buttons, etc.) are represented with respect to previous widget or parent View. A Relative Layout example is shown in the figure below.

Using RelativeLayout, we can align two elements by right border, or make one below another, centered in the screen, centered left, and so on. By default, all child views are drawn at the top-left of the layout, so we must define the position of each view using the various layout properties available from **RelativeLayout.LayoutParams**.

Example:

```

<RelativeLayout
    android:id="@+id/RLayout"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"

```

```
xmlns:android="http://schemas.android.com/apk/res/android" >
</RelativeLayout>
```

3. List View Layout:

A List View is a View Layout where all the items are specified in the form of a list. Android **List View** is a view which groups several items and display them in vertical scrollable list. The list items are automatically inserted to the list using an **Adapter** that pulls content from a source such as an array or database.

An adapter actually bridges between UI components and the data source that fill data into UI Component. Adapter holds the data and send the data to adapter view, the view can take the data from adapter view and shows the data on different views like as spinner, list view, grid view etc.

The **List View** and **GridView** are subclasses of **AdapterView** and they can be populated by binding them to an **Adapter**, which retrieves data from an external source and creates a View that represents each data entry.

4. GridView Layout:

A GridView is a View Layout where the items (such as pictures, files etc.) are placed in a Grid manner. Android **GridView** shows items in two-dimensional scrolling grid (rows & columns) and the grid items are not necessarily predetermined but they automatically inserted to the layout using a **ListAdapter**.

5. TableLayout:

Android TableLayout going to be arranged groups of views into rows and columns. We use the <TableRow> element to build a row in the table. Each row has zero or more cells; each cell can hold one View object.

Example:

```
<TableLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_height="fill_parent"
android:layout_width="fill_parent" >

<TableRow>
<TextView
    android:text="User Name:"
    android:width="120dp"
    />

<EditText
    android:id="@+id/txtUserName"
    android:width="200dp" />
</TableRow>
</TableLayout>
```

6. Absolute Layout:

An Absolute Layout lets us specify exact locations (x/y coordinates) of its children. Absolute layouts are less flexible and harder to maintain than other types of layouts without absolute positioning.

Example:

```
<AbsoluteLayout
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    xmlns:android="http://schemas.android.com/apk/res/android" >

    <Button
        android:layout_width="188dp"
        android:layout_height="wrap_content"
        android:text="Button"
        android:layout_x="126px"
        android:layout_y="361px" />
</AbsoluteLayout>
```

7. FrameLayout:

The FrameLayout is a placeholder on screen that we can use to display a single view.

Example:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignLeft="@+id/lblComments"
    android:layout_below="@+id/lblComments"
    android:layout_centerHorizontal="true" >

    <ImageView
        android:src="@drawable/droid"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</FrameLayout>
```

8. ConstraintLayout:

The fundamental building block of ConstraintLayout is creating constraints. A constraint defines a relationship between two widgets in the layout and controls how those widgets will be positioned within the layout

Resources:

Resources are the additional files and static content that our code uses, such as bitmaps, layout definitions, user interface strings and animation instructions. The Android

resource system keeps track of all non-code assets associated with an application. We can use this class to access our application's resources. We can generally acquire the Resources instance associated with our application with getResources().

For any type of resource, we can specify *default* and multiple *alternative* resources for our application:

- ❖ Default resources are those that should be used regardless of the device configuration or when there are no alternative resources that match the current configuration.
- ❖ Alternative resources are those that you've designed for use with a specific configuration. To specify that a group of resources are for a specific configuration, append an appropriate configuration qualifier to the directory name.

Directory and Resources Type:

❖ anim/

XML files that define property animations. They are saved in res/anim/ folder and accessed from the R.anim class.

❖ color/

XML files that define a state list of colors. They are saved in res/color/ and accessed from the R.color class.

❖ drawable/

Image files like .png, .jpg, .gif or XML files that are compiled into bitmaps, state lists, shapes, animation drawable. They are saved in res/drawable/ and accessed from the R.drawable class.

❖ layout/

XML files that define a user interface layout. They are saved in res/layout/ and accessed from the R.layout class.

❖ menu/

XML files that define application menus, such as an Options Menu, Context Menu, or Sub Menu. They are saved in res/menu/ and accessed from the R.menu class.

❖ raw/

Arbitrary files to save in their raw form. We need to call Resources.openRawResource() with the resource ID, which is R.raw.filename to open such raw files.

❖ values/

XML files that contain simple values, such as strings, integers, and colors. For example, here are some filename conventions for resources we can create in this directory:

- arrays.xml for resource arrays, and accessed from the R.array class.
- integers.xml for resource integers, and accessed from the R.integer class.

- bools.xml for resource boolean, and accessed from the R.bool class.
- colors.xml for color values, and accessed from the R.color class.
- dimens.xml for dimension values, and accessed from the R.dimen class.
- strings.xml for string values, and accessed from the R.string class.
- styles.xml for styles, and accessed from the R.style class.

❖ xml/

Arbitrary XML files that can be read at runtime by calling `Resources.getXML()`. We can save various configuration files here which will be used at run time.

Android Style:

A **style** is a collection of properties that specify the look and format for a View or window. A style can specify properties such as height, padding, font color, font size, background color, and much more. A style is defined in an XML resource that is separate from the XML that specifies the layout.

Styles in Android share a similar philosophy to cascading stylesheets in web design they allow us to separate the design from the content.

For example, by using a style, we can take this layout XML:

```
<TextView
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:textColor="#00FF00"
android:typeface="monospace"
android:text="@string/hello" />
```

A **theme** is a style applied to an entire Activity or application, rather than an individual View (as in the example above). When a style is applied as a theme, every View in the Activity or application will apply each style property that it supports.

To create a set of styles, save an XML file in the `res/values/` directory of android project. The name of the XML file is arbitrary, but it must use the `.xml` extension and be saved in the `res/values/` folder.

The root node of the XML file must be `<resources>`.

For each style we want to create, add a `<style>` element to the file with a name that uniquely identifies the style (this attribute is required). Then add an `<item>` element for each property of that style, with a name that declares the style property and a value to go with it (this attribute is required). The value for the `<item>` can be a keyword string, a hex color, a reference to another resource type, or other value depending on the style property.

Here's an example file with a single style:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
```

```
<style name="CodeFont" parent="@android:style/TextAppearance.Medium">
  <item name="android:layout_width">fill_parent</item>
  <item name="android:layout_height">wrap_content</item>
  <item name="android:textColor">#00FF00</item>
  <item name="android:typeface">monospace</item>
</style>
</resources>
```

ANDROID THIRD PARTY UI/UX LIBRARIES:

Google wants Android to be in everything that has web connectivity and to that effect, the company has unveiled what third party apps might one day look like inside our car. Android for the car will be called Android Auto and the image we see here is what the apps might one day look like in the car, assuming our car supports Android Auto.

Google did the same thing with Android Wear where the apps for Wear aren't complete apps, rather they are used to control what content from an app on our smartphone we can see on the wearable device.

There are many third-party libraries for Android but several of them are "must have" libraries that are extremely popular and are often used in almost any Android project. Each has different purposes but all of them make life as a developer much more pleasant.

Some 3rd Party UI/UX Libraries:

❖ ActionBarSherlock:

ActionBarSherlock is an extension of the support library designed to facilitate the use of the action bar design pattern across all versions of Android with a single API. ActionBarSherlock was widely used before Google introduced AppCompatActivity.

The library will automatically use the native action bar when appropriate or will automatically wrap a custom implementation around our layouts. This allows us to easily develop an application with an action bar for every version of Android from 2.x and up.

❖ ActionBarPullToRefresh:








ActionBar PullToRefresh provides an easy way to add a modern version of the pull-to-refresh interaction to our application. ActionBar-PullToRefresh has in-built support for:

- AbsListView derivatives (ListView & GridView).
- ScrollView
- WebView

❖ Android PullToRefresh:

This project aims to provide a reusable Pull to Refresh widget for Android. It was originally based on Johan Nilsson's library (mainly for graphics, strings and animations), but these have been replaced since.

Features:

- Supports both Pulling Down from the top, and Pulling Up from the bottom (or even both).
- Animated Scrolling for all devices.
- Over Scroll supports for devices on Android v2.3+.
- Currently works with:
 -  **ListView**
 -  **ExpandableListView**
 -  **GridView**
 -  **WebView**
 -  **ScrollView**
 -  **HorizontalScrollView**
 -  **ViewPager**
- Integrated End of List Listener for use of detecting when the user has scrolled to the bottom.
- Maven Support.
- Indicators to show the user when a Pull-to-Refresh is available.
- Support for **ListFragment**!

❖ **View Pager Indicator:**

Paging indicator widgets compatible with the ViewPager from the Android Support Library and ActionBarSherlock.

❖ **Nine Old Android:**

Android library for using the Honeycomb (Android 3.0) animation API on all versions of the platform back to 1.0! Animation prior to Honeycomb was very limited in what it could accomplish so in Android 3.x a new API was written

❖ **A Chart Engine:**

A ChartEngine is a charting library for Android applications. It currently supports all major and widely used chart types.

❖ **ProgressWheel:**

A progress wheel is used to indicate an app is busy loading content to display. It can be applied to a complete screen, when the whole page has to be loaded, or only to a part of the page, e.g. in a module tab in which content has to be loaded. In addition, a small progress wheel in the title bar or action bar can be added.

The progress wheel is displayed within the current screen. So not on top of it in a dialog, nor is it triggered by a specific action from the user, like the **progress wheel dialog** in both cases would be. It is shown e.g. when the user moves from one app, screen, or tab to the other and the page needs some time to load the content. The progress wheel doesn't quantify the progress that's being made. For that you should use a progress bar.

❖ **Segmented Radio Buttons:**

For text-only buttons, we just need `SegmentedRadioGroup.java` which extends `RadioGroup`, so all our standard `RadioButton` implementations and callbacks should work.

For image buttons, implement `SegmentedRadioImageButton` instead of `RadioButton`.

Drawables are included, but can easily be replaced.