COMP-304
Fall 2018

Review of Lecture 2

- **Android Activities**:
  - **Activity:** a task in an Android application
  - Extends class Activity
  - Life Cycle:
    - onCreate
    - onStart
    - onRestart
    - onResume
    - onPause
    - onStop
    - onDestroy
- **Android Manifest File** - application configuration information
  - Application's package name
  - The components of the app, which include all activities, services, broadcast receivers, and content providers
    - The permissions
    - Hardware and software features
- Some attributes are defined in build.gradle file

Review of Lecture 2

- **Registering Activities**:

```
<activity android:name=".AudioActivity" />
```

  - Primary Point Activity:

```
<intent-filter>
<action
    android:name="android.intent.action.MAIN" />
<category
    android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

- **Registering permissions:**

```
<uses-permission
    android:name="android.permission.CAMERA" />
```

- **Managing Resources**
  - /res folder
  - Layout, drawables, values
  - R.java class
  - **findViewById**(R.id.myimageview)

- **Intents**
  - Used to call other activities and built-in apps
  - startActivity method
    - Source activity
    - Started activity
  - Intent objects
    - the action to be performed
    - the data to be acted upon
  - Passing information
    - putExtra
    - getExtras

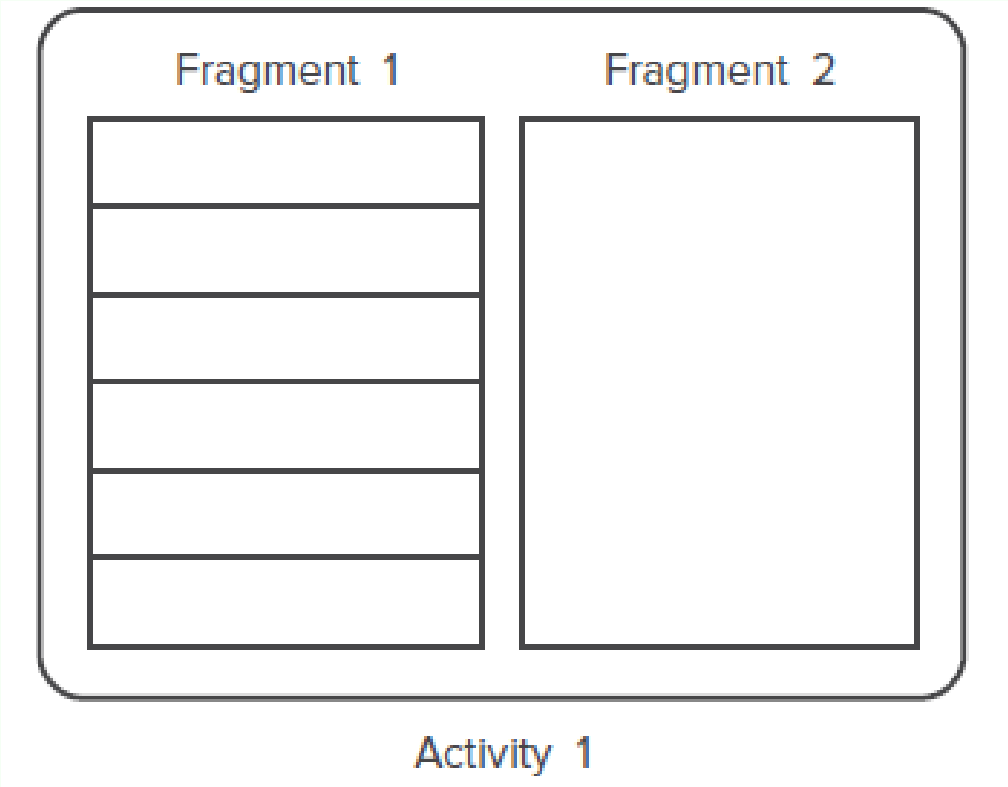
Lecture 3

Objectives:

- Explain and use Fragments
- Manage Application resources:
  - Declare simple resource values - Strings, Integers, Booleans, Colors, Drawables, String Arrays, XML files, etc
  - Simple Resources Example
- Create and use Android User Interface elements
  - Explain Android Layout classes
  - Explain and use Simple UI controls and event handling:
    - TextView
    - EditText
    - Button

FRAGMENTS

- A fragment is a **mini activity**.
- An activity can have many fragments to contain views.



FRAGMENTS

- Fragments are Java classes and load their UIs from corresponding XML files
- A fragment extends the Fragment base class:

```
public class Fragment1 extends Fragment {
}
```

- To add a fragment to an activity, you use the <fragment> element

```
<?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="horizontal" >
<fragment
android:name="net.learn2develop.Fragments.Fragment1"
android:id="@+id/fragment1"
android:layout_weight="1"
android:layout_width="0px"
android:layout_height="match_parent" />
```

FRAGMENTS

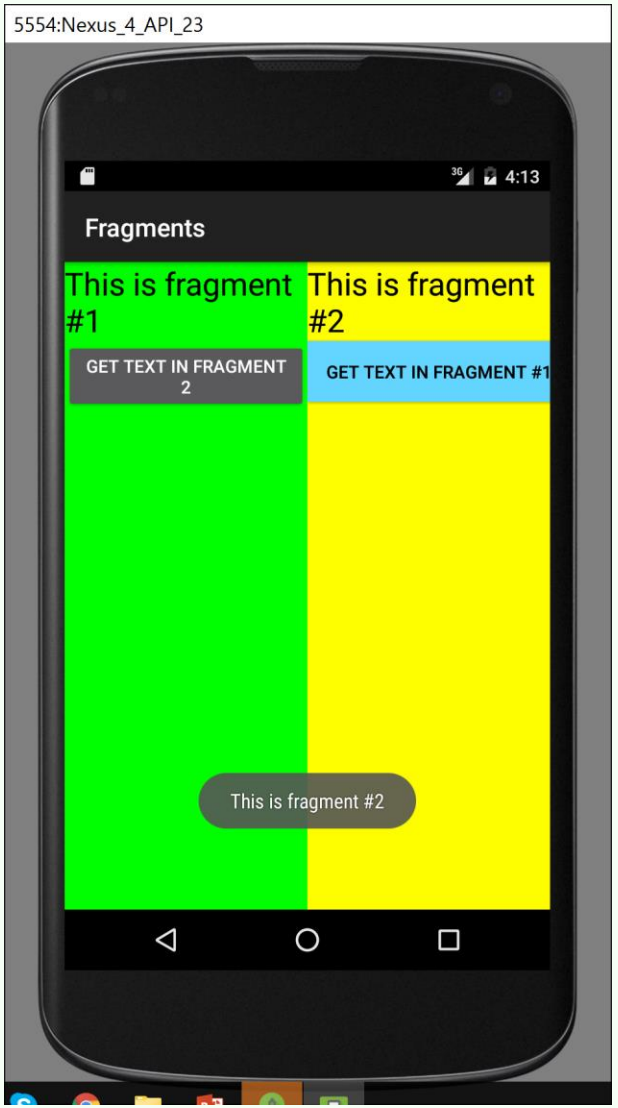
- You can add fragments dynamically to activities during runtime.
  - FragmentManager class
  - FragmentTransaction class

```
FragmentManager fragmentManager = getFragmentManager();
FragmentTransaction fragmentTransaction =
fragmentManager.beginTransaction();
//---get the current display info---
WindowManager wm = getWindowManager();
Display d = wm.getDefaultDisplay();
if (d.getWidth() > d.getHeight())
{
        //---landscape mode---
        Fragment1 fragment1 = new Fragment1();
        // android.R.id.content refers to the content view of the activity
        fragmentTransaction.replace(
        android.R.id.content, fragment1);
}
```

FRAGMENTS

```
else
{
//---portrait mode---
Fragment2 fragment2 = new Fragment2();
fragmentTransaction.replace(
android.R.id.content, fragment2);
}
fragmentTransaction.commit();
```

- FragmentsInteractionActivity example



Fragment's life cycle

- When a fragment is being created, it goes through the following states:
  - **onAttach()** - fragment is associated with its activity
  - **onCreate()** - initial creation of the fragment
  - **onCreateView()** – creates fragment view
  - **onActivityCreated()** – its activity finished onCreate
- When the fragment becomes visible, it goes through these states:
  - **onStart()** - makes the fragment visible to the user
  - **onResume()** - makes the fragment interacting with the user

Fragment's life cycle

- When the fragment goes into the background mode, it goes through these states:
  - onPause()
  - onStop()
- When the fragment is destroyed (when the activity it is currently hosted in is destroyed), it goes through the following states:
  - onPause() - fragment is no longer interacting with the user
  - onStop() - fragment is no longer visible to the user
  - onDestroyView() - clean up resources associated with its View
  - onDestroy() - final cleanup of the fragment's state
  - onDetach() – before the fragment no longer being associated with its activity

many of the life cycle evts are similar to Activities

only that fragments have even more life cycle events

Fragment's life cycle

- Like activities, you can restore an instance of a fragment using a Bundle object, in the following states:
  - onCreate()
  - onCreateView()
  - onActivityCreated()

Interactions between Fragments

- Example:
  - you can obtain the activity in which a fragment is currently embedded by first using the **getActivity()** method and then using the **findViewById()** method to locate the view(s) contained within the fragment:

```
TextView lbl = (TextView)
getActivity().findViewById(R.id.lblFragment1);
Toast.makeText(getActivity(), lbl.getText(),
Toast.LENGTH_SHORT).show();
```

Setting Simple Resource Values

- You can define resource types by **editing resource XML files manually** or by **using resource editors** available in Android Studio.
  - Here is a view of /res/values/strings.xml file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Using Android Resources</string>
    <string name="display">Demonstrating Font and Color!</string>
    <string name="app_name">Simple Resource Example</string>
    <color name="prettyTextColor">#fa31ff</color>
    <dimen name="textPointSize">14pt</dimen>
    <drawable name="redDrawable">#ff0000</drawable>
</resources>
```

Setting Simple Resource Values

- You can create:
  - A **String** resource named *hello* with a value of "Using Android Resources"
  - A **String** resource named *display* with a value of "Demonstrating Font and Color!"
  - A **color** resource named *prettyTextColor* with a value of #ff0000
  - A **dimension** resource named *textPointSize* with a value of 14pt
  - A **drawable** resource named *redDrawable* with a value of #F00

Setting Simple Resource Values

❑ The generated R.java file:

```
package test.simpleresources;
public final class R {
    public static final class attr {
    }
    public static final class color {
        public static final int prettyTextColor=0x7f050000;
    }
    public static final class dimen {
        public static final int textPointSize=0x7f060000;
    }
    public static final class drawable {
        public static final int icon=0x7f020000;
        public static final int redDrawable=0x7f020001;
    }
    public static final class layout {
        public static final int main=0x7f030000;
    }
    public static final class string {
        public static final int app_name=0x7f040001;
        public static final int hello=0x7f040000;
    }
}
```

Accessing resources from Java code

```
package test.simpleresources;

import android.app.Activity;
import android.graphics.drawable.ColorDrawable;
import android.os.Bundle;
import android.widget.ImageView;
import android.widget.TextView;

public class SimpleResource extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
```

Accessing resources from Java code

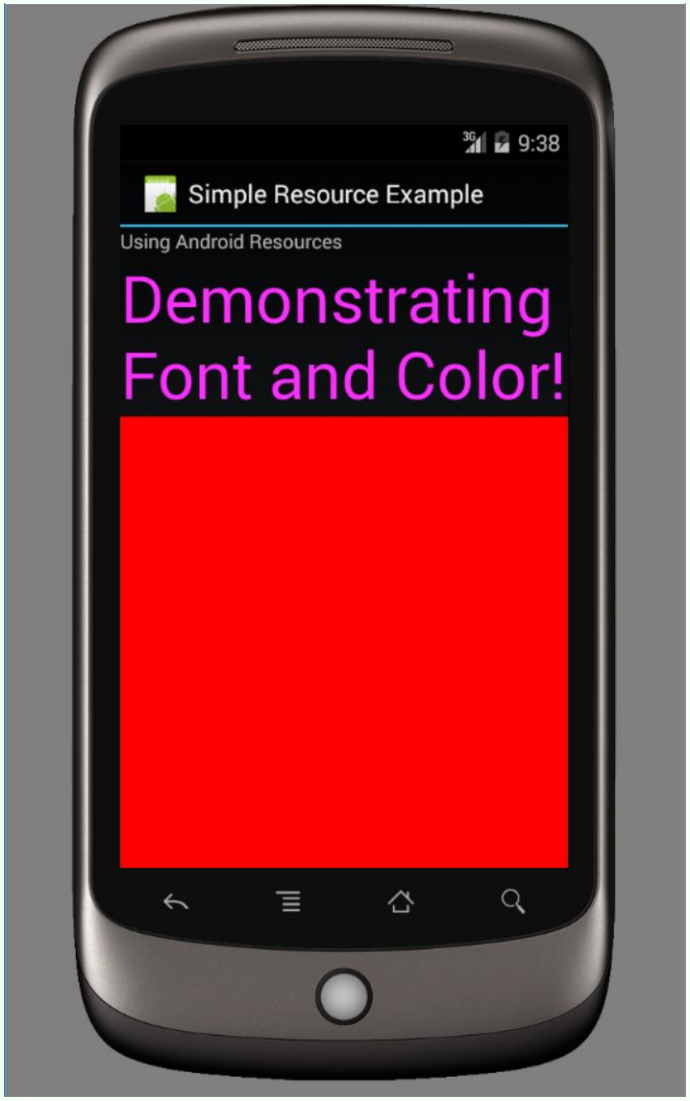
```
        String myString = getResources().getString(R.string.display);
        int myColor = getResources().getColor(R.color.prettyTextColor);
        float myDimen = getResources().getDimension(R.dimen.textPointSize);
        ColorDrawable myDraw =
      (ColorDrawable)getResources().getDrawable(R.drawable.redDrawable);
        ImageView imgView = (ImageView)findViewById(R.id.imageView1);

        imgView.setImageDrawable(myDraw);
        //String[] flavors = getResources().getStringArray(R.array.flavors);

        TextView tv = (TextView)findViewById(R.id.txtView);
        tv.setTextSize(myDimen);
        tv.setTextColor(myColor);
        tv.setText(myString);

    }
}
```

Simple Resources example



Defining String Arrays

- String arrays, may be added to resource files by editing them manually:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Hello World, SimpleResource!</string>
    <string name="app_name">Simple Resource Example</string>
    <color name="prettyTextColor">#ff0000</color>
    <dimen name="textPointSize">14pt</dimen>
    <drawable name="redDrawable">#F00</drawable>
    <string-array name="flavors">
        <item>Vanilla</item>
        <item>Chocolate</item>
        <item>Strawberry</item>
    </string-array>
</resources>
```

- Access the array in your code:

```
String[] aFlavors = getResources().getStringArray(R.array.flavors);
```

Working with Resources

- It is a common practice to store different types of resources in different files.
- For example, store:
  - the strings in **/res/values/strings.xml**
  - the prettyTextColor color resource in **/res/values/colors.xml**
  - the textPointSize dimension resource in **/res/values/dimens.xml**
- This does not change the names of the resources, nor the code used earlier to access the resources programmatically

it's basically like a dictionary/properties file => rsrcs stored as key value pairs with type being the tag

Working with Boolean Resources

- **Boolean** resources are defined in XML under the /res/values project directory and compiled into the application package at build time
- Are tagged with the <bool> tag and represent a name-value pair:

```
<resources>
    <bool name="bOnePlusOneEqualsTwo">true</bool>
    <bool name="bAdvancedFeaturesEnabled">false</bool>
</resources>
```

- The following code retrieves a boolean resource named bAdvancedFeaturesEnabled:

```
boolean bAdvancedMode =
getResources().getBoolean(R.bool.bAdvancedFeaturesEnabled);
```

Working with Integer Resources

- **Integer** values are tagged with the <integer> tag and represent a name/value pair.

```
<resources>
    <integer name="numTimesToRepeat">25</integer>
    <integer name="startingAgeOfCharacter">3</integer>
</resources>
```

- The following code accesses your application's integer resource named numTimesToRepeat:

```
int repTimes =
    getResources().getInteger(R.integer.numTimesToRepeat);
```

Working with Colors

- Android applications can store RGB color values, which can then be applied to other screen elements
- The following color formats are supported:
  - #RGB (example, #F00 is 12-bit color, red)
  - #ARGB (example, #8F00 is 12-bit color, red with alpha 50%)
  - #RRGGBB (example, #FF00FF is 24-bit color, magenta)
  - #AARRGGBB (example, #80FF00FF is 24-bit color, magenta with alpha 50%)
- Color values are tagged with the <color> tag and represent a name-value pair:

```
<resources>
    <color name=”background_color”>#006400</color>
    <color name=”text_color”>#FFE4C4</color>
</resources>
```

- The following code retrieves a color resource called prettyTextColor:

```
int myResourceColor = getResources().getColor(R.color.prettyTextColor);
```

Working with Dimensions

- Many user interface layout controls such as text controls and buttons are drawn to specific dimensions.
  - These dimensions can be stored as resources.
- Dimension values always end with a unit of measurement tag:

Pixels	Actual screen pixels	**px**	20px
Inches	Physical measurement	**in**	1in
Millimeters	Physical measurement	**mm**	1mm
Points	Common font measurement unit	**pt**	14pt
Screen density Independent Pixels	Pixels relative to 160dpi screen (preferable dimension for screen compatibility). One **dp** is one pixel on a 160 dpi screen. dp = (width in pixels * 160) / screen density	**dp**	1dp
Scale independent Pixels	Best for scalable font display sp preserves a user's font settings	**sp**	14sp

- Dimension values are tagged with the <dimen> tag and represent a name/value pair.

Working with Dimensions

- Here's an example of a simple dimension resource file /res/values/dimens.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <dimen name="FourteenPt">14pt</dimen>
    <dimen name="OneInch">1in</dimen>
    <dimen name="TenMillimeters">10mm</dimen>
    <dimen name="TenPixels">10px</dimen>
</resources>
```

- Dimension resources are simply floating point values.
  - The following code retrieves a dimension resource called textPointSize:

```
float myDimension =
getResources().getDimension(R.dimen.textPointSize);
```

Working with Simple Drawables

- **Simple paintable drawable** resources are defined in XML under the /res/values project directory and compiled into the application package at build time.
- **Paintable drawable resources** use the <drawable> tag and represent a name-value pair.
  - Here's an example of a simple drawable resource file /res/values/drawables.xml:

```
<resources>
    <drawable name="red_rect">#F00</drawable>
</resources>
```

- Drawable resources defined with <drawable> are simply **rectangles of a given color**:

```
ColorDrawable myDraw = (ColorDrawable)getResources().
getDrawable(R.drawable.redDrawable);
```

Using Image Resources Programmatically

- Images resources are simply **another kind of Drawable** called a **BitmapDrawable**
- Use resource ID of the image to set as an attribute on a user interface control:

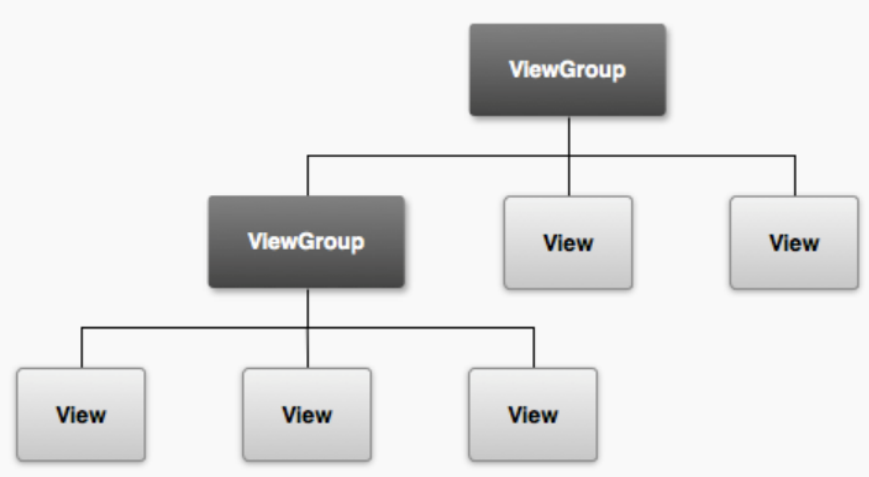
```
ImageView flagImageView =
(ImageView)findViewById(R.id.ImageView01);
flagImageView.setImageResource(R.drawable.flag);
```

- You can access the BitmapDrawable object directly:

```
BitmapDrawable bitmapFlag = (BitmapDrawable)
getResources().getDrawable(R.drawable.flag);
```

Android Views

- android.**view** package contains a number of interfaces and classes related to drawing on the screen.
- android.view.**View** class is the basic user interface building block within Android.
  - It represents a rectangular portion of the screen.
- The View class serves as the **base class** for nearly all the user interface controls and layouts within the Android SDK.
- ViewGroup is an invisible container that defines the layout structure for View and other ViewGroup object



Creating Layouts Using XML Resources

- Layouts and user interface controls can be **defined as application resources** or **created programmatically** at runtime
- Android provides a simple way to create layout files in XML as resources provided in the **/res/layout** project directory
  - This is the most common way

Layout classes

- LinearLayout
- AbsoluteLayout
- TableLayout
- RelativeLayout
- ConstraintLayout
- FrameLayout
- ScrollView

- AbsoluteLayout may be used to specify the exact x/y coordinate locations of each control on the screen instead, but this is **not easily portable across many screen resolutions**

Creating Layouts Using XML Resources

- You can configure almost any ViewGroup or View (or View subclass) attribute using the XML layout resource files
- **LinearLayout** is the **default layout** file provided with any new Android project in Android Studio, referred to as /res/layout/main.xml:

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

Creating Layouts Using XML Resources

- **LinearLayout** is a ViewGroup that shows each child View either in a **single column** or in a **single row**.
- When applied to a full screen, each child View is drawn under the previous View if the **orientation** is set to **vertical** or to the right of the previous View if orientation is set to **horizontal**
- To associate the **main.xml** layout with the activity, use the method call **setContentView()** with the identifier of the **main.xml** layout.
  - The ID of the layout matches the XML filename without the extension:

setContentView(R.layout.main);

Creating Layouts Using XML Resources

```
<?xml version=”1.0” encoding=”utf-8”?>
<LinearLayout xmlns:android=
“http://schemas.android.com/apk/res/android”
android:orientation=”vertical”
android:layout_width=”fill_parent”
android:layout_height=”fill_parent”
>
<TextView
android:id=”@+id/TextView1”
android:layout_width=”fill_parent”
android:layout_height=”wrap_content”
android:text=”Hi There!”
/>
<TextView
android:id=”@+id/TextView2”
android:layout_width=”fill_parent”
android:layout_height=”wrap_content”
android:textSize=”60px”
android:text=”I’m second. I need to wrap.”
/>
</LinearLayout>
```

Creating Layouts Programmatically

- You can create user interface components such as layouts at runtime programmatically:

```
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView text1 = new TextView(this);
        text1.setText("Hi there!");
        TextView text2 = new TextView(this);
        text2.setText("I'm second. I need to wrap.");
        text2.setTextSize((float) 60);
        LinearLayout ll = new LinearLayout(this);
        ll.setOrientation(LinearLayout.VERTICAL);
        ll.addView(text1);
        ll.addView(text2);
        setContentView(ll);
}
```

Organizing Your User Interface

- You add child View objects to a **ViewGroup** programmatically using the method addView().
- In XML, you add child objects to a ViewGroup by defining the child View control as a child node in the XML
- **ViewGroup** subclasses are broken down into two categories:
  - **Layout** classes
  - View **container controls**

Using Built-In Layout Classes

- All layouts, regardless of their type, have basic layout attributes:

  android:layout_attribute_name=”value”

- Common attributes include the size attributes and margin attributes

Attribute Name	Applies To	Description	Value
`android:layout_height`	Parent view Child view	Height of the view. Required attribute for child view controls in layouts.	Specific dimension value, `fill_parent`, or `wrap_content`. The `match_parent` option is available in API Level 8+.
`android:layout_width`	Parent view Child view	Width of the view. Required attribute for child view controls in layouts.	Specific dimension value, `fill_parent`, or `wrap_content`. The `match_parent` option is available in API Level 8+.
`android:layout_margin`	Child view	Extra space on all sides of the view.	Specific dimension value.

ViewGroup Attributes

- This example of a LinearLayout sets the size of the screen, containing one TextView that is set to its full **height** and the **width** of the LinearLayout (and therefore the screen):

```
<LinearLayout xmlns:android=
“http://schemas.android.com/apk/res/android”
android:layout_width=”fill_parent”
android:layout_height=”fill_parent”>
<TextView
android:id=”@+id/TextView01”
android:layout_height=”fill_parent”
android:layout_width=”fill_parent” />
</LinearLayout>
```

ViewGroup Attributes

- Here is an example of a **Button** object with some margins set via XML used in a layout resource file:

```
<Button
android:id=”@+id/Button01”
android:layout_width=”wrap_content”
android:layout_height=”wrap_content”
android:text=”Press Me”
android:layout_marginRight=”20px”
android:layout_marginTop=”60px” />
```

Using **FrameLayout**

- **FrameLayout** view is designed to display **a stack of child View items**.
- You can add multiple views to this layout, but **each View is drawn from the top-left corner** of the layout.
- Use this to show multiple images within the same region, and the **layout is sized to the largest child View** in the stack.

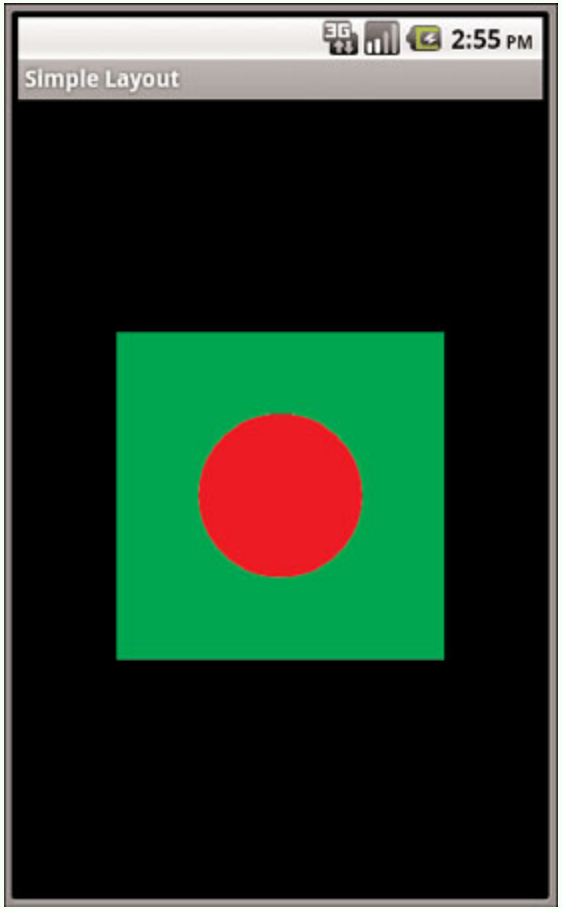
Using **FrameLayout**

- Here's an example of an XML layout resource with a FrameLayout and **two child View objects**, both **ImageView** objects.
- The green rectangle is drawn first and the red oval is drawn on top of it.
- The green rectangle is larger, so it defines the bounds of the FrameLayout:

```
<FrameLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
android:id="@+id/FrameLayout01"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_gravity="center">
```

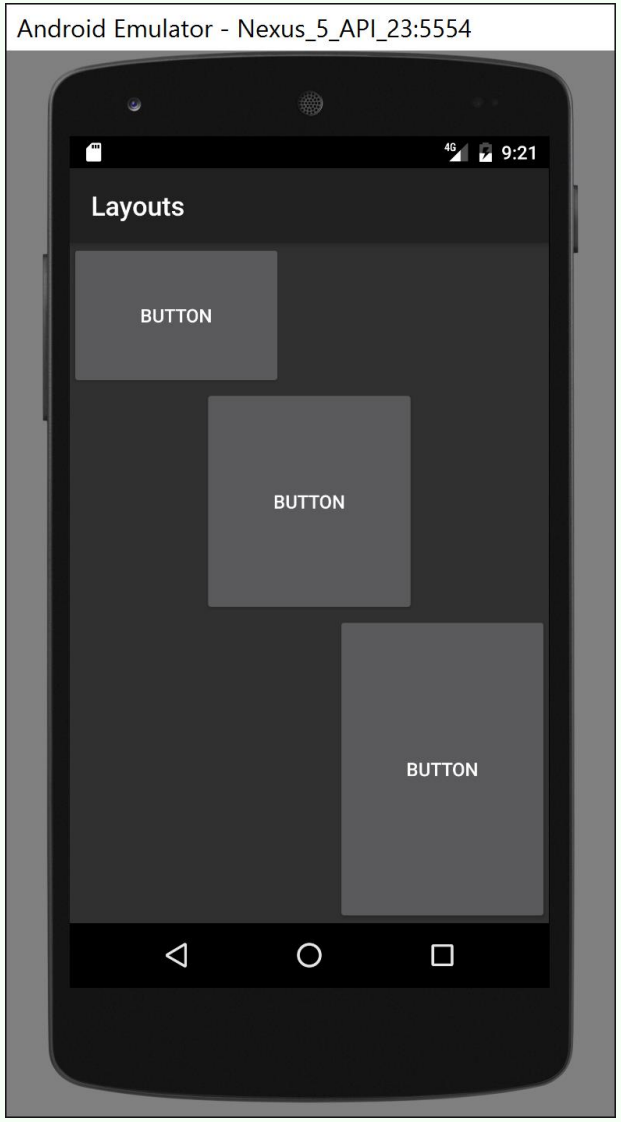
Using **FrameLayout**

```
<ImageView
android:id=”@+id/ImageView01”
android:layout_width=”wrap_content”
android:layout_height=”wrap_content”
android:src=”@drawable/green_rect”
android:minHeight=”200px”
android:minWidth=”200px” />
<ImageView
android:id=”@+id/ImageView02”
android:layout_width=”wrap_content”
android:layout_height=”wrap_content”
android:src=”@drawable/red_oval”
android:minHeight=”100px”
android:minWidth=”100px”
android:layout_gravity=”center” />
</FrameLayout>
```



Using **LinearLayout**

- A LinearLayout view organizes its child View objects in a single row, or column, depending on whether its orientation attribute is set to horizontal or vertical.
- Very handy layout method for creating forms



Using **RelativeLayout**

- The **RelativeLayout** view enables you to specify where the child view controls are **in relation to each other**.
- For instance, you can set a child View to be positioned “above” or “below” or “to the left of ” or “to the right of ” another View, referred to by its unique identifier.
- You can also align child View objects relative to one another or the parent layout edges.

Using **RelativeLayout**

- Combining RelativeLayout attributes can simplify creating interesting user interfaces without resorting to multiple layout groups to achieve a desired effect.
- The picture shows how each of the button controls is relative to each other

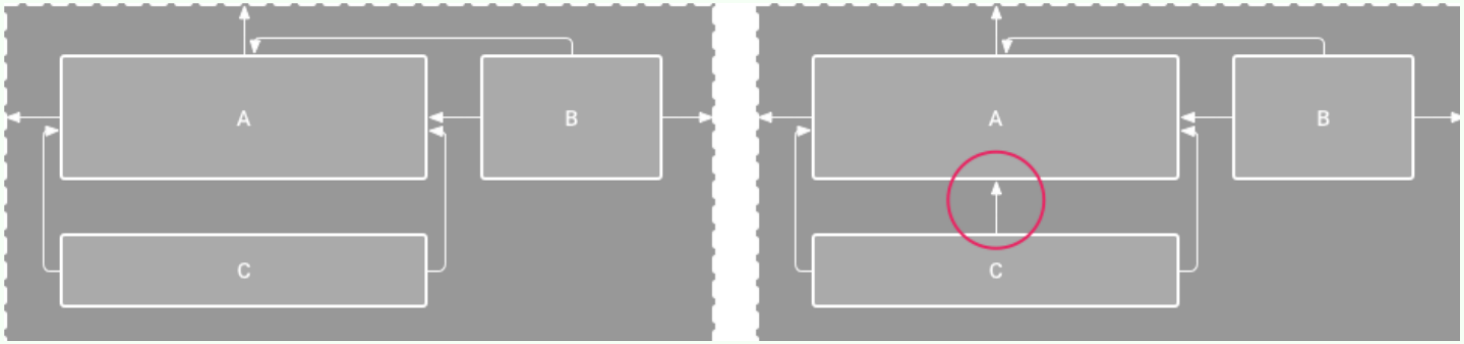
Using **RelativeLayout**

- Here's an example of an XML layout resource with a RelativeLayout and two child View objects, a Button object aligned relative to its parent, and an ImageView aligned and positioned relative to the Button (and the parent):

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
android:id="@+id/RelativeLayout01"
android:layout_height="fill_parent"
android:layout_width="fill_parent">
<Button
android:id="@+id/ButtonCenter"
android:text="Center"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_centerInParent="true" />
<ImageView
android:id="@+id/ImageView01"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_above="@id/ButtonCenter"
android:layout_centerHorizontal="true"
android:src="@drawable/arrow" />
</RelativeLayout>
```

ConstraintLayout

- ConstraintLayout allows you to create large and complex layouts with a flat view hierarchy (no nested view groups).
- It's similar to RelativeLayout in that **all views are laid out according to relationships between sibling views and the parent layout**, but it's more flexible than RelativeLayout and easier to use with Android Studio's Layout Editor.



- The editor shows view C below A, but it has no vertical constraint
- View C is now **vertically constrained below view A**

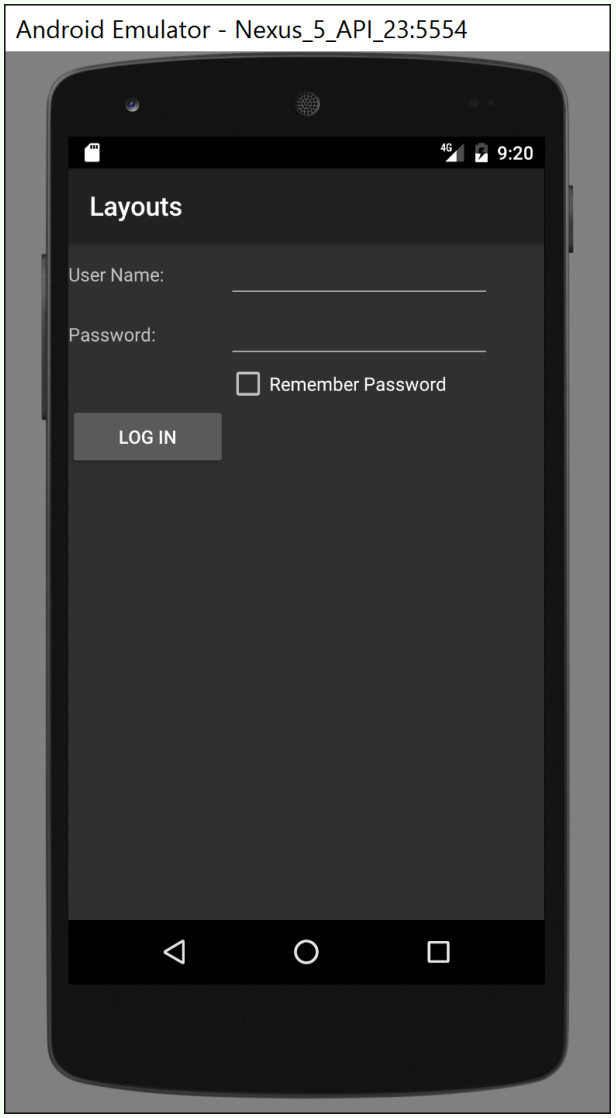
https://developer.android.com/training/constraint-layout/

Using **TableLayout**

- A **TableLayout** view organizes children into rows,
- You add individual View objects within each row of the table using a **TableRow** layout View (which is basically a horizontally oriented LinearLayout) for each row of the table.
  - Each **column** of the TableRow can contain one View (or layout with child View objects).
  - You place View items added to a TableRow in columns in the order they are added.
  - You can specify the **column number** (zero-based) to skip columns as necessary; otherwise, the View object is put in the next column to the right.
  - **Columns scale** to the size of the largest View of that column.

Using **TableLayout**

- You can also include normal View objects instead of TableRow elements, if you want the View to take up an entire row.



Using **TableLayout**

- Here's an example of an XML layout resource with a TableLayout with two rows (two TableRow child objects).
- The TableLayout is set to **stretch the columns** to the size of the screen width.
  - The first TableRow has three columns; each cell has a Button object.
  - The second TableRow puts only one Button view into the second column explicitly:

```
<TableLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
android:id="@+id/TableLayout01"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:stretchColumns="*">
<TableRow
android:id="@+id/TableRow01">
<Button
android:id="@+id/ButtonLeft"
android:text="Left Door" />
```

Using **TableLayout**

```
<Button
android:id="@+id/ButtonMiddle"
android:text="Middle Door" />
<Button
android:id="@+id/ButtonRight"
android:text="Right Door" />
</TableRow>
<TableRow
android:id="@+id/TableRow02">
<Button
android:id="@+id/ButtonBack"
android:text="Go Back"
android:layout_column="1" />
</TableRow>
</TableLayout>
```

Using Multiple Layouts on a Screen

- Combining different layout methods on a single screen can create complex layouts.
- Because a layout contains View objects and is, itself, a View, it can contain other layouts.
- The figure on the right demonstrates a combination of layout views used in conjunction to create a more complex and interesting screen

Layout example

ScrollView

- A ScrollView is a special type of FrameLayout in that it enables users to **scroll through a list of views** that occupy more space than the physical display.
- The ScrollView can **contain only one child** view or ViewGroup, which normally is a LinearLayout.

```
<ScrollView
android:layout_width=”fill_parent”
android:layout_height=”fill_parent”
xmlns:android=”http://schemas.android.com/apk/res/android” >
    <LinearLayout
        android:layout_width=”fill_parent”
        android:layout_height=”wrap_content”
        android:orientation=”vertical” >
        ……..
    </LinearLayout>
</ScrollView>
```

Android Controls

- android.**widget** contains Android controls
- The Android SDK includes classes to draw most common objects, including **ImageView**, **FrameLayout**, **EditText**, and **Button** classes.
  - All controls are typically **derived from the View** class

Displaying Text to Users with TextView

- TextView control is used to draw text on the screen.
  - You primarily use it to **display fixed text** strings or labels
- It is derived from View and is within the **android.widget** package:
  - all the standard attributes such as width, height, padding, and visibility can be applied to the object
- You can set the **android:text** property of the TextView to be either a raw text string in the layout file or a reference to a string resource.

Displaying Text to Users with TextView

```
<TextView android:id=”@+id/TextView01”
android:layout_width=”wrap_content”
android:layout_height=”wrap_content”
android:text=”Some sample text here” />

<TextView android:id=”@+id/TextView02”
android:layout_width=”wrap_content”
android:layout_height=”wrap_content”
android:text=”@string/sample_text” />
```

Displaying Text to Users with **TextView**

- Call **setContentView**() method with the layout resource identifier to display this TextView on the screen

```
displayTextView = (TextView) findViewById(R.id.TextView01);
```

- You can change the text displayed programmatically by calling the **setText()** method on the TextView object.
- Retrieving the text is done with the **getText()** method.

Retrieving Data from Users

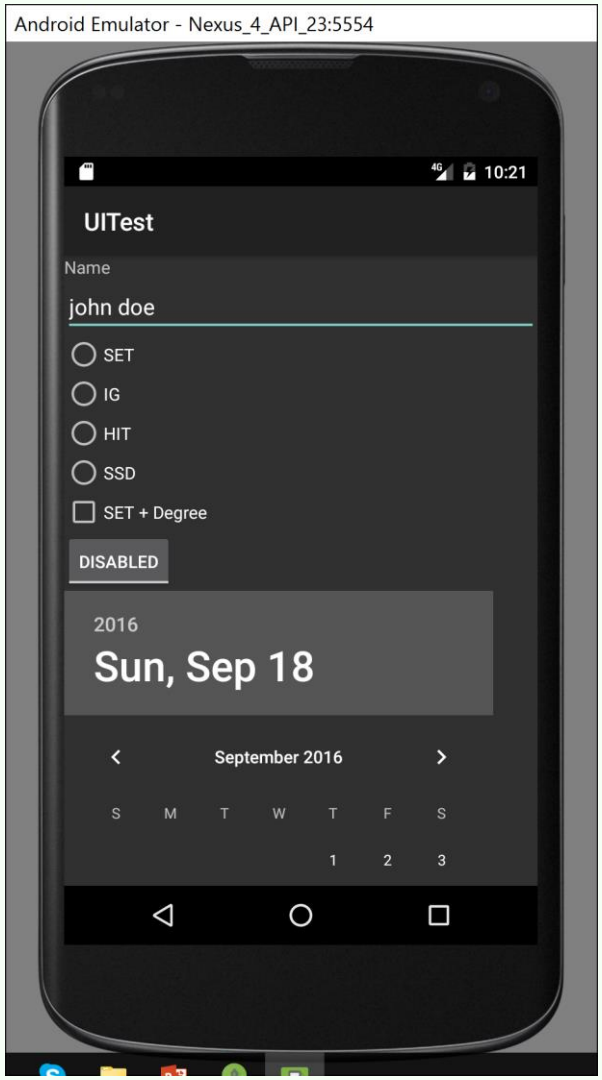
- Two frequently used controls to handle this type of job are **EditText** controls and **Spinner** controls
- **EditText** handles text input from a user.
  - The EditText class is **derived from TextView**
- This is how to define an EditText control in an XML layout file:

```
<EditText
android:id="@+id/EditText01"
android:layout_height="wrap_content"
android:hint="type here"
android:lines="4"
android:layout_width="fill_parent" />
```

- **hint** attribute gives a hint to the user as to what should be typed in EditText control
- **lines** attribute, which defines how many lines tall the input box is

Retrieving Data from Users

- To highlight a portion of the text from code use **setSelection()** method, and a call to **selectAll()** highlights the entire text entry field.
- EditText object is essentially an editable TextView.
  - This means that you can read text from it in the same way as you did with TextView: by using the **getText()** method.
  - You can also set initial text to draw in the text entry area using the **setText()** method



Using Buttons

- The android.widget.**Button** class provides a basic button implementation in the Android SDK.
- Within the XML layout resources, buttons are specified using the Button element.
- The primary attribute for a basic button is the text field
- Use basic Button controls for buttons with text such as “Ok,””Cancel,” or “Submit.”

Using Buttons

- The following XML layout resource file shows a typical Button control definition:

```
<Button
android:id="@+id/basic_button"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Basic Button" />
```

- This is some code that handles a click for a basic button and displays a Toast message on the screen:

```
final Button basic_button = (Button) findViewById(R.id.basic_button);
basic_button.setOnClickListener(new View.OnClickListener() {
public void onClick(View v) {
Toast.makeText(ButtonsActivity.this, "Button clicked",
    Toast.LENGTH_SHORT).show();
}
});
```

References

- Textbook
- Android Documentation
- https://material.io/guidelines/layout/units-measurements.html
- https://developer.android.com/guide/topics/ui/overview.html
- https://developer.android.com/guide/topics/ui/declaring-layout.html
- Lauren Darcey, Shane Conder: Introduction to Android Application Development: Android Essentials (5th Edition)