

## MODULE – 2

### Android – Resources

There are many more items which you use to build a good Android application. Apart from coding for the application, you take care of various other **resources** like static content that your code uses, such as bitmaps, colors, layout definitions, user interface strings, animation instructions, and more. These resources are always maintained separately in various sub-directories under **res/** directory of the project.

### Organize resource in Android Studio

```
MyProject/  
  app/  
    manifest/  
      AndroidManifest.xml  
  java/  
    MainActivity.java  
  res/  
    drawable/  
      icon.png  
    layout/  
      activity_main.xml  
      info.xml  
    values/  
      strings.xml
```

Sl. No.	Directory & Resource Type
1	<b>anim/</b> XML files that define property animations. They are saved in res/anim/ folder and accessed from the <b>R.anim</b> class.
2	<b>color/</b> XML files that define a state list of colors. They are saved in res/color/ and accessed from the <b>R.color</b> class.

3	<p><b>drawable/</b></p> <p>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 <b>R.drawable</b> class.</p>
4	<p><b>layout/</b></p> <p>XML files that define a user interface layout. They are saved in res/layout/ and accessed from the <b>R.layout</b> class.</p>
5	<p><b>menu/</b></p> <p>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 <b>R.menu</b> class.</p>
6	<p><b>raw/</b></p> <p>Arbitrary files to save in their raw form. You need to call <i>Resources.openRawResource()</i> with the resource ID, which is <i>R.raw.filename</i> to open such raw files.</p>
7	<p><b>values/</b></p> <p>XML files that contain simple values, such as strings, integers, and colors. For example, here are some filename conventions for resources you can create in this directory –</p> <ul style="list-style-type: none"><li>• arrays.xml for resource arrays, and accessed from the <b>R.array</b> class.</li><li>• integers.xml for resource integers, and accessed from the <b>R.integer</b> class.</li><li>• bools.xml for resource boolean, and accessed from the <b>R.bool</b> class.</li><li>• colors.xml for color values, and accessed from the <b>R.color</b> class.</li><li>• dimens.xml for dimension values, and accessed from the <b>R.dimen</b> class.</li><li>• strings.xml for string values, and accessed from the <b>R.string</b> class.</li><li>• styles.xml for styles, and accessed from the <b>R.style</b> class.</li></ul>
8	<p><b>xml/</b></p> <p>Arbitrary XML files that can be read at runtime by calling <i>Resources.getXML()</i>. You can save various configuration files here which will be used at run time.</p>

## Alternative Resources

Your application should provide alternative resources to support specific device configurations. For example, you should include alternative drawable resources ( i.e.images ) for different screen resolution and alternative string resources for different languages. At runtime, Android detects the current device configuration and loads the appropriate resources for your application.

To specify configuration-specific alternatives for a set of resources, follow the following steps –

- Create a new directory in res/ named in the form **<resources\_name>-<config\_qualifier>**. Here **resources\_name** will be any of the resources mentioned in the above table, like layout, drawable etc. The **qualifier** will specify an individual configuration for which these resources are to be used. You can check official documentation for a complete list of qualifiers for different type of resources.
- Save the respective alternative resources in this new directory. The resource files must be named exactly the same as the default resource files as shown in the below example, but these files will have content specific to the alternative. For example though image file name will be same but for high resolution screen, its resolution will be high.

Below is an example which specifies images for a default screen and alternative images for high resolution screen.

```
MyProject/  
  app/  
    manifest/  
      AndroidManifest.xml  
  java/  
    MyActivity.java  
  res/  
    drawable/  
      icon.png  
      background.png  
    drawable-hdpi/  
      icon.png  
      background.png  
  layout/  
    activity_main.xml  
    info.xml  
  values/  
    strings.xml
```

## Accessing Resources

During your application development you will need to access defined resources either in your code, or in your layout XML files. Following section explains how to access your resources in both the scenarios –

### Accessing Resources in Code

When Android application is compiled, a **R** class gets generated, which contains resource IDs for all the resources available in your **res/** directory. You can use R class to access that resource using sub-directory and resource name or directly resource ID.

#### Example

To access *res/drawable/myimage.png* and set an ImageView you will use following code –

```
ImageView imageView = (ImageView) findViewById(R.id.myimageview);  
imageView.setImageResource(R.drawable.myimage);
```

Here first line of the code make use of *R.id.myimageview* to get ImageView defined with id *myimageview* in a Layout file. Second line of code makes use of *R.drawable.myimage* to get an image with name **myimage** available in drawable sub-directory under **/res**.

#### Example

Consider next example where *res/values/strings.xml* has following definition –

```
<?xml version="1.0" encoding="utf-8"?>  
<resources>  
  <string name="hello">Hello, World!</string>  
</resources>
```

Now you can set the text on a TextView object with ID msg using a resource ID as follows –

```
TextView msgTextView = (TextView) findViewById(R.id.msg);  
msgTextView.setText(R.string.hello);
```

#### Example

Consider a layout *res/layout/activity\_main.xml* with the following definition –

```
<?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:id="@+id/text"  
    android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:text="Hello, I am a TextView" />

        <Button android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello, I am a Button" />

    </LinearLayout>
```

This application code will load this layout for an Activity, in the onCreate() method as follows –

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
}
```

### Accessing Resources in XML

Consider the following resource XML *res/values/strings.xml* file that includes a color resource and a string resource –

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="opaque_red">#f00</color>
    <string name="hello">Hello!</string>
</resources>
```

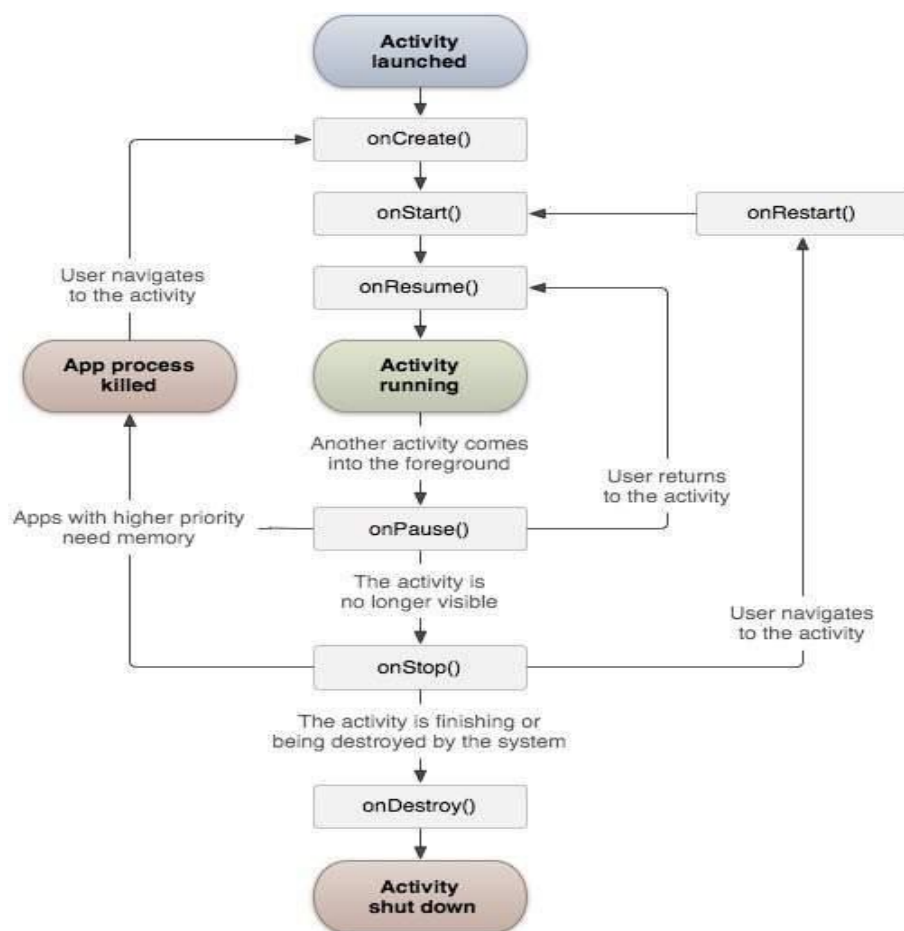
Now you can use these resources in the following layout file to set the text color and text string as follows –

```
<?xml version="1.0" encoding="utf-8"?>
<EditText xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:textColor="@color/opaque_red"
    android:text="@string/hello" />
```

## Android – Activities

An activity represents a single screen with a user interface just like window or frame of Java.

**Activity Life Cycle:** There is a sequence of callback methods that start up an activity and a sequence of callback methods that tear down an activity.

**Activity life cycle diagram**

The Activity class defines the following call backs i.e. events. You don't need to implement all the callbacks methods. However, it's important that you understand each one and implement those that ensure your app behaves the way users expect.

Sr.No	Callback & Description
1	<b>onCreate():</b> This is the first callback and called when the activity is first created.
2	<b>onStart():</b> This callback is called when the activity becomes visible to the user.
3	<b>onResume():</b> This is called when the user starts interacting with the application.

4	<b>onPause():</b> The paused activity does not receive user input and cannot execute any code and called when the current activity is being paused and the previous activity is being resumed.
5	<b>onStop():</b> This callback is called when the activity is no longer visible.
6	<b>onDestroy():</b> This callback is called before the activity is destroyed by the system.
7	<b>onRestart():</b> This callback is called when the activity restarts after stopping it.

### Logs in android.

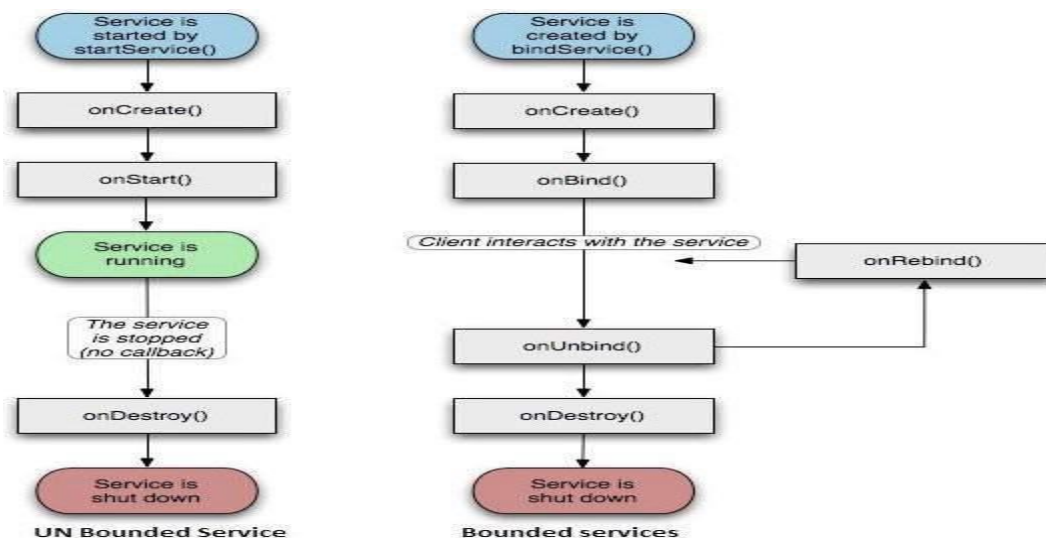
- ☐ Log.e: This is for when bad stuff happens. Use this tag in places like inside a catch statement. You *know* that an *error* has occurred and therefore you're logging an error.
- ☐ Log.w: Use this when you suspect something shady is going on. You may not be completely in full on error mode, but maybe you recovered from some unexpected behavior. Basically, use this to log stuff you didn't expect to happen but isn't necessarily an error. Kind of like a "hey, this happened, and it's *weird*, we should look into it."
- ☐ Log.i: Use this to post useful *information* to the log. For example: that you have successfully connected to a server. Basically, use it to report successes.
- ☐ Log.d: Use this for *debugging* purposes. If you want to print out a bunch of messages so you can log the exact flow of your program, use this. If you want to keep a log of variable values, use this.
- ☐ Log.v: Use this when you want to go absolutely nuts with your logging. If for some reason you've decided to log every little thing in a particular part of your app, use the Log.v tag.

### Android – Services

A **service** is a component that runs in the background to perform long-running operations without needing to interact with the user and it works even if application is destroyed. A service can essentially take two states –

Sl.No.	State & Description
1	<p><b>Started</b></p> <p>A service is <b>started</b> when an application component, such as an activity, starts it by calling <code>startService()</code>. Once started, a service can run in the background indefinitely, even if the component that started it is destroyed.</p>
2	<p><b>Bound</b></p> <p>A service is <b>bound</b> when an application component binds to it by calling <code>bindService()</code>. A bound service offers a client-server interface that allows components to interact with the service, send requests, get results, and even do so across processes with interprocess communication (IPC).</p>

A service has life cycle callback methods that you can implement to monitor changes in the service's state and you can perform work at the appropriate stage. The following diagram on the left shows the life cycle when the service is created with `startService()` and the diagram on the right shows the life cycle when the service is created with `bindService()`: (image courtesy : [android.com](http://android.com) )





To create an service, you create a Java class that extends the Service base class or one of its existing subclasses. The **Service** base class defines various callback methods and the most important are given below. You don't need to implement all the callbacks methods. However, it's important that you understand each one and implement those that ensure your app behaves the way users expect.

Sl.No.	Callback & Description
1	<p><b>onStartCommand()</b></p> <p>The system calls this method when another component, such as an activity, requests that the service be started, by calling <i>startService()</i>. If you implement this method, it is your responsibility to stop the service when its work is done, by calling <i>stopSelf()</i> or <i>stopService()</i> methods.</p>
2	<p><b>onBind()</b></p> <p>The system calls this method when another component wants to bind with the service by calling <i>bindService()</i>. If you implement this method, you must provide an interface that clients use to communicate with the service, by returning an <i>IBinder</i> object. You must always implement this method, but if you don't want to allow binding, then you should return <i>null</i>.</p>
3	<p><b>onUnbind()</b></p> <p>The system calls this method when all clients have disconnected from a particular interface published by the service.</p>
4	<p><b>onRebind()</b></p> <p>The system calls this method when new clients have connected to the service, after it had previously been notified that all had disconnected in its <i>onUnbind(Intent)</i>.</p>
5	<p><b>onCreate()</b></p> <p>The system calls this method when the service is first created using <i>onStartCommand()</i> or <i>onBind()</i>. This call is required to perform one-time set-up.</p>
6	<p><b>onDestroy()</b></p>

The system calls this method when the service is no longer used and is being destroyed. Your service should implement this to clean up any resources such as threads, registered listeners, receivers, etc.

### Android – Broadcast Receiver

**Broadcast Receivers** simply respond to broadcast messages from other applications or from the system itself. These messages are sometime called events or intents. For example, applications can also initiate broadcasts to let other applications know that some data has been downloaded to the device and is available for them to use, so this is broadcast receiver who will intercept this communication and will initiate appropriate action.

There are following two important steps to make `BroadcastReceiver` works for the system broadcasted intents –

- Creating the Broadcast Receiver.
- Registering Broadcast Receiver

There is one additional steps in case you are going to implement your custom intents then you will have to create and broadcast those intents.

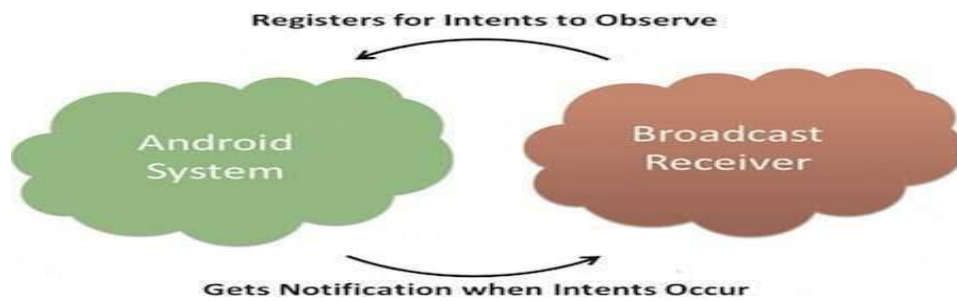
#### Creating the Broadcast Receiver

A broadcast receiver is implemented as a subclass of **`BroadcastReceiver`** class and overriding the `onReceive()` method where each message is received as a **`Intent`** object parameter.

```
public class MyReceiver extends BroadcastReceiver {  
    @Override  
    public void onReceive(Context context, Intent intent) {  
        Toast.makeText(context, "Intent Detected.", Toast.LENGTH_LONG).show();  
    }  
}
```

#### Registering Broadcast Receiver

An application listens for specific broadcast intents by registering a broadcast receiver in *AndroidManifest.xml* file. Consider we are going to register *MyReceiver* for system generated event `ACTION_BOOT_COMPLETED` which is fired by the system once the Android system has completed the boot process.



```

<application
  android:icon="@drawable/ic_launcher"
  android:label="@string/app_name"
  android:theme="@style/AppTheme" >
  <receiver android:name="MyReceiver">

    <intent-filter>
      <action android:name="android.intent.action.BOOT_COMPLETED">
    </action>
    </intent-filter>

  </receiver>
</application>
  
```

Now whenever your Android device gets booted, it will be intercepted by BroadcastReceiver *MyReceiver* and implemented logic inside *onReceive()* will be executed.

There are several system generated events defined as final static fields in the **Intent** class. The following table lists a few important system events.

Sr.No	Event Constant & Description
1	<b>android.intent.action.BATTERY_CHANGED</b> : Sticky broadcast containing the charging state, level, and other information about the battery.
2	<b>android.intent.action.BATTERY_LOW</b> : Indicates low battery condition on the device.
3	<b>android.intent.action.BATTERY_OKAY</b> : Indicates the battery is now okay after being low.

4	<b>android.intent.action.BOOT_COMPLETED</b> : This is broadcast once, after the system has finished booting.
5	<b>android.intent.action.BUG_REPORT</b> : Show activity for reporting a bug.
6	<b>android.intent.action.CALL</b> : Perform a call to someone specified by the data.
7	<b>android.intent.action.CALL_BUTTON</b> : The user pressed the "call" button to go to the dialer or other appropriate UI for placing a call.
8	<b>android.intent.action.DATE_CHANGED</b> : The date has changed.
9	<b>android.intent.action.REBOOT</b> : Have the device reboot.

### Broadcasting Custom Intents

If you want your application itself should generate and send custom intents then you will have to create and send those intents by using the `sendBroadcast()` method inside your activity class. If you use the `sendStickyBroadcast(Intent)` method, the Intent is **sticky**, meaning the *Intent* you are sending stays around after the broadcast is complete.

```
public void broadcastIntent(View view) {  
    Intent intent = new Intent();  
    intent.setAction("com.sdmc.CUSTOM_INTENT");  
    sendBroadcast(intent);  
}
```

This intent `com.sdmc.CUSTOM_INTENT` can also be registered in similar way as we have registered system generated intent.

```
<application  
    android:icon="@drawable/ic_launcher"  
    android:label="@string/app_name"  
    android:theme="@style/AppTheme" >  
    <receiver android:name="MyReceiver">  
  
        <intent-filter>
```

```

        <action android:name="com.sdmc.CUSTOM_INTENT">
        </action>
    </intent-filter>

    </receiver>
</application>

```

### Android – Content Providers

A content provider component supplies data from one application to others on request. Such requests are handled by the methods of the `ContentResolver` class. A content provider can use different ways to store its data and the data can be stored in a database, in files, or even over a network.



Content providers let you centralize content in one place and have many different applications access it as needed. A content provider behaves very much like a database where you can query it, edit its content, as well as add or delete content using `insert()`, `update()`, `delete()`, and `query()` methods. In most cases this data is stored in an **SQLite** database.

A content provider is implemented as a subclass of **ContentProvider** class and must implement a standard set of APIs that enable other applications to perform transactions.

```

public class My Application extends ContentProvider {
}

```

### Content URIs

To query a content provider, you specify the query string in the form of a URI which has following format –

```

<prefix>://<authority>/<data_type>/<id>

```

Here is the detail of various parts of the URI –

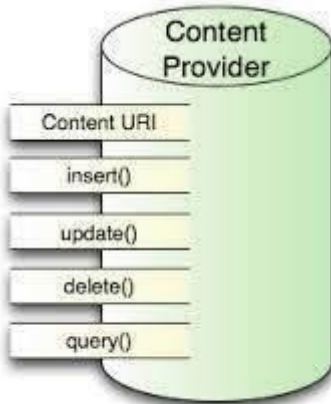
Sr.No	Part & Description
1	<b>prefix:</b> This is always set to content://
2	<b>authority:</b> This specifies the name of the content provider, for example <i>contacts</i> , <i>browser</i> etc. For third-party content providers, this could be the fully qualified name, such as <i>com.tutorialspoint.statusprovider</i>
3	<b>data_type:</b> This indicates the type of data that this particular provider provides. For example, if you are getting all the contacts from the <i>Contacts</i> content provider, then the data path would be <i>people</i> and URI would look like this <i>content://contacts/people</i>
4	<b>id:</b> This specifies the specific record requested. For example, if you are looking for contact number 5 in the <i>Contacts</i> content provider then URI would look like this <i>content://contacts/people/5</i> .

### Create Content Provider

This involves number of simple steps to create your own content provider.

- First of all you need to create a Content Provider class that extends the *ContentProviderbase* class.
- Second, you need to define your content provider URI address which will be used to access the content.
- Next you will need to create your own database to keep the content. Usually, Android uses SQLite database and framework needs to override *onCreate()* method which will use SQLite Open Helper method to create or open the provider's database. When your application is launched, the *onCreate()* handler of each of its Content Providers is called on the main application thread.
- Next you will have to implement Content Provider queries to perform different database specific operations.
- Finally register your Content Provider in your activity file using <provider> tag.

Here is the list of methods which you need to override in Content Provider class to have your Content Provider working –



- **onCreate()** This method is called when the provider is started.
- **query()** This method receives a request from a client. The result is returned as a Cursor object.
- **insert()** This method inserts a new record into the content provider.
- **delete()** This method deletes an existing record from the content provider.
- **update()** This method updates an existing record from the content provider.
- **getType()** This method returns the MIME type of the data at the given URI.

### Android – Fragments

A **Fragment** is a piece of an activity which enable more modular activity design. It will not be wrong if we say, a fragment is a kind of **sub-activity**.

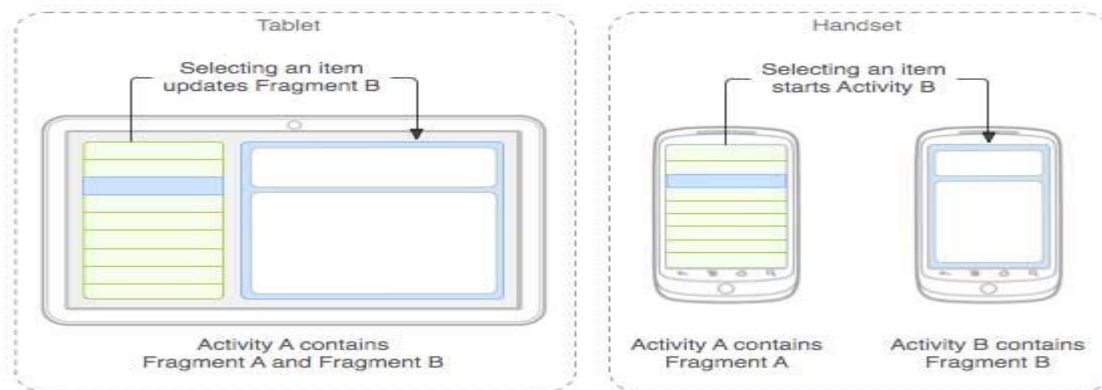
Following are important points about fragment –

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

You create fragments by extending **Fragment** class and you can insert a fragment into your activity layout by declaring the fragment in the activity's layout file, as a **<fragment>** element.

Prior to fragment introduction, we had a limitation because we can show only a single activity on the screen at one given point in time. So we were not able to divide device screen and control different parts separately. But with the introduction of fragment we got more flexibility and removed the limitation of having a single activity on the screen at a time. Now we can have a single activity but each activity can comprise of multiple fragments which will have their own layout, events and complete life cycle.

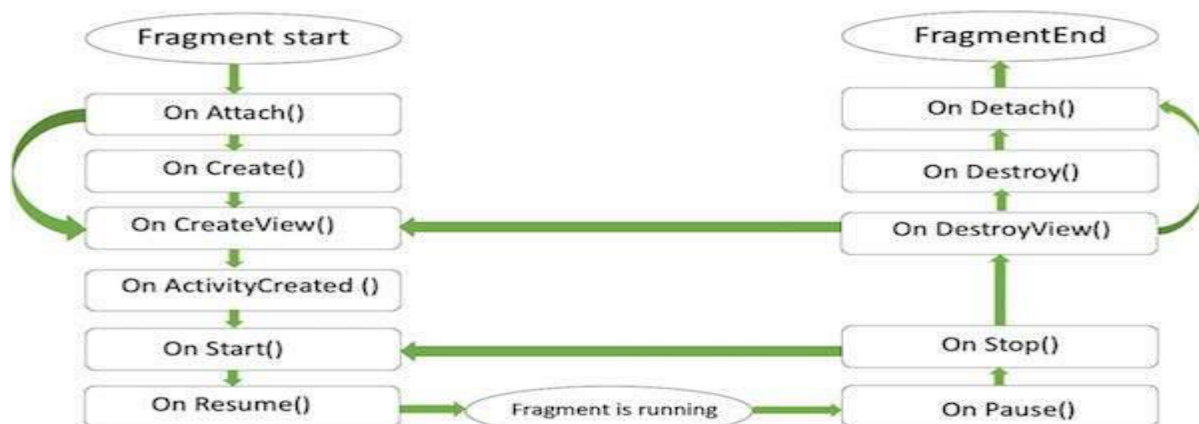
Following is a typical example of how two UI modules defined by fragments can be combined into one activity for a tablet design, but separated for a handset design.



The application can embed two fragments in Activity A, when running on a tablet-sized device. However, on a handset-sized screen, there's not enough room for both fragments, so Activity A includes only the fragment for the list of articles, and when the user selects an article, it starts Activity B, which includes the second fragment to read the article.

### Fragment Life Cycle

Android fragments have their own life cycle very similar to an android activity. This section briefs different stages of its life cycle.





Here is the list of methods which you can to override in your fragment class –

- **onAttach()** The fragment instance is associated with an activity instance. The fragment and the activity is not fully initialized. Typically you get in this method a reference to the activity which uses the fragment for further initialization work.
- **onCreate()** The system calls this method when creating the fragment. You should initialize essential components of the fragment that you want to retain when the fragment is paused or stopped, then resumed.
- **onCreateView()** The system calls this callback when it's time for the fragment to draw its user interface for the first time. To draw a UI for your fragment, you must return a **View** component from this method that is the root of your fragment's layout. You can return null if the fragment does not provide a UI.
- **onActivityCreated()** The **onActivityCreated()** is called after the **onCreateView()** method when the host activity is created. Activity and fragment instance have been created as well as the view hierarchy of the activity. At this point, view can be accessed with the **findViewById()** method. example. In this method you can instantiate objects which require a Context object
- **onStart()** The **onStart()** method is called once the fragment gets visible.
- **onResume()** Fragment becomes active.
- **onPause()** The system calls this method as the first indication that the user is leaving the fragment. This is usually where you should commit any changes that should be persisted beyond the current user session.
- **onStop()** Fragment going to be stopped by calling **onStop()**
- **onDestroyView()** Fragment view will destroy after call this method
- **onDestroy()** **onDestroy()** called to do final clean up of the fragment's state but Not guaranteed to be called by the Android platform.

### How to use Fragments?

This involves number of simple steps to create Fragments.

- First of all decide how many fragments you want to use in an activity. For example let's we want to use two fragments to handle landscape and portrait modes of the device.

- Next based on number of fragments, create classes which will extend the *Fragment* class. The *Fragment* class has above mentioned callback functions. You can override any of the functions based on your requirements.
- Corresponding to each fragment, you will need to create layout files in XML file. These files will have layout for the defined fragments.
- Finally modify activity file to define the actual logic of replacing fragments based on your requirement.

### Types of Fragments

Basically fragments are divided as three stages as shown below.

- Single frame fragments – Single frame fragments are using for hand hold devices like mobiles, here we can show only one fragment as a view.
- List fragments – fragments having special list view is called as list fragment
- Fragments transaction – Using with fragment transaction. we can move one fragment to another fragment.

### Android : Intents and Filters

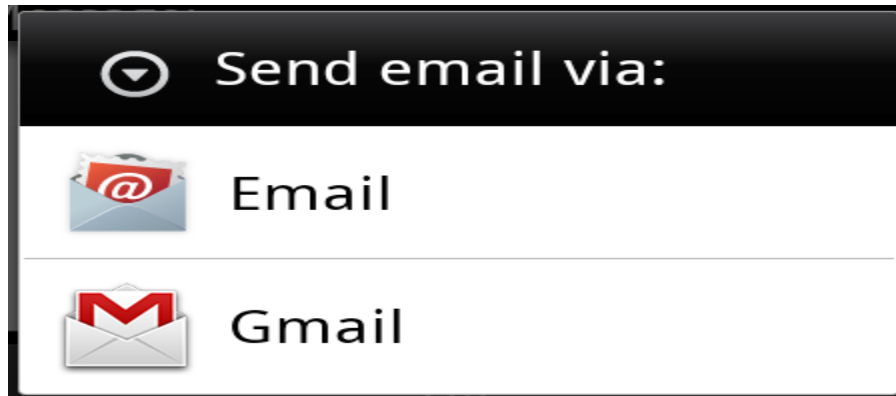
An Android **Intent** is an abstract description of an operation to be performed. It can be used with **startActivity** to launch an Activity, **broadcastIntent** to send it to any interested BroadcastReceiver components, and **startService(Intent)** or **bindService(Intent, ServiceConnection, int)** to communicate with a background Service.

**The intent itself, an Intent object, is a passive data structure holding an abstract description of an operation to be performed.**

For example, let's assume that you have an Activity that needs to launch an email client and sends an email using your Android device. For this purpose, your Activity would send an ACTION\_SEND along with appropriate **chooser**, to the Android Intent Resolver. The specified chooser gives the proper interface for the user to pick how to send your email data.

```
Intent email = new Intent(Intent.ACTION_SEND, Uri.parse("mailto:"));
email.putExtra(Intent.EXTRA_EMAIL, recipients);
email.putExtra(Intent.EXTRA_SUBJECT, subject.getText().toString());
email.putExtra(Intent.EXTRA_TEXT, body.getText().toString());
startActivity(Intent.createChooser(email, "Choose an email client from..."));
```

Above syntax is calling startActivity method to start an email activity and result should be as shown below –



For example, assume that you have an Activity that needs to open URL in a web browser on your Android device. For this purpose, your Activity will send ACTION\_WEB\_SEARCH Intent to the Android Intent Resolver to open given URL in the web browser. The Intent Resolver parses through a list of Activities and chooses the one that would best match your Intent, in this case, the Web Browser Activity. The Intent Resolver then passes your web page to the web browser and starts the Web Browser Activity.

```
String q = "tutorialspoint";
Intent intent = new Intent(Intent.ACTION_WEB_SEARCH);
intent.putExtra(SearchManager.QUERY, q);
startActivity(intent);
```

Above example will search as **tutorialspoint** on android search engine and it gives the result of tutorialspoint in your an activity

There are separate mechanisms for delivering intents to each type of component – activities, services, and broadcast receivers.

Sr.No	Method & Description
1	<b>Context.startActivity():</b> The Intent object is passed to this method to launch a new activity or get an existing activity to do something new.
2	<b>Context.startService():</b> The Intent object is passed to this method to initiate a service or deliver new instructions to an ongoing service.
3	<b>Context.sendBroadcast():</b> The Intent object is passed to this method to deliver the message

	to all interested broadcast receivers.
--	--

## Intent Objects

An Intent object is a bundle of information which is used by the component that receives the intent as well as information used by the Android system.

An Intent object can contain the following components based on what it is communicating or going to perform –

### Action

This is mandatory part of the Intent object and is a string naming the action to be performed — or, in the case of broadcast intents, the action that took place and is being reported. The action largely determines how the rest of the intent object is structured . The Intent class defines a number of action constants corresponding to different intents. Here is a list of [Android Intent Standard Actions](#)

The action in an Intent object can be set by the `setAction()` method and read by `getAction()`.

### Data

Adds a data specification to an intent filter. The specification can be just a data type (the `contentType` attribute), just a URI, or both a data type and a URI. A URI is specified by separate attributes for each of its parts –

These attributes that specify the URL format are optional, but also mutually dependent –

- If a scheme is not specified for the intent filter, all the other URI attributes are ignored.
- If a host is not specified for the filter, the port attribute and all the path attributes are ignored.

The `setData()` method specifies data only as a URI, `setType()` specifies it only as a MIME type, and `setDataAndType()` specifies it as both a URI and a MIME type. The URI is read by `getData()` and the type by `getType()`.

Some examples of action/data pairs are –

Sr.No.	Action/Data Pair & Description
1	<b>ACTION_VIEW</b> <code>content://contacts/people/1</code> : Display information about the person whose identifier is "1".

2	<b>ACTION_DIAL</b> content://contacts/people/1: Display the phone dialer with the person filled in.
3	<b>ACTION_VIEW</b> <a href="#">tel:123</a> : Display the phone dialer with the given number filled in.
4	<b>ACTION_DIAL</b> <a href="#">tel:123</a> : Display the phone dialer with the given number filled in.
5	<b>ACTION_EDIT</b> content://contacts/people/1: Edit information about the person whose identifier is "1".
6	<b>ACTION_VIEW</b> content://contacts/people/: Display a list of people, which the user can browse through.
7	<b>ACTION_SET_WALLPAPER</b> : Show settings for choosing wallpaper
8	<b>ACTION_SYNC</b> : It going to be synchronous the data, Constant Value is <b>android.intent.action.SYNC</b>
9	<b>ACTION_SYSTEM_TUTORIAL</b> : It will start the platform-defined tutorial(Default tutorial or start up tutorial)
10	<b>ACTION_TIMEZONE_CHANGED</b> : It intimates when time zone has changed
11	<b>ACTION_UNINSTALL_PACKAGE</b> : It is used to run default uninstaller

### Category

The category is an optional part of Intent object and it's a string containing additional information about the kind of component that should handle the intent. The addCategory() method places a category in an Intent object, removeCategory() deletes a category previously added, and getCategories() gets the set of all categories currently in the object. Here is a list of Android Intent Standard Categories.

You can check detail on Intent Filters in below section to understand how do we use categories to choose appropriate activity corresponding to an Intent.

**Extras**

This will be in key-value pairs for additional information that should be delivered to the component handling the intent. The extras can be set and read using the `putExtras()` and `getExtras()` methods respectively. Here is a list of [Android Intent Standard Extra Data](#)

**Flags**

These flags are optional part of Intent object and instruct the Android system how to launch an activity, and how to treat it after it's launched etc.

Sr.No	Flags & Description
1	<b>FLAG_ACTIVITY_CLEAR_TASK</b> If set in an Intent passed to <code>Context.startActivity()</code> , this flag will cause any existing task that would be associated with the activity to be cleared before the activity is started. That is, the activity becomes the new root of an otherwise empty task, and any old activities are finished. This can only be used in conjunction with <code>FLAG_ACTIVITY_NEW_TASK</code> .
2	<b>FLAG_ACTIVITY_CLEAR_TOP</b> If set, and the activity being launched is already running in the current task, then instead of launching a new instance of that activity, all of the other activities on top of it will be closed and this Intent will be delivered to the (now on top) old activity as a new Intent.
3	<b>FLAG_ACTIVITY_NEW_TASK</b> This flag is generally used by activities that want to present a "launcher" style behavior: they give the user a list of separate things that can be done, which otherwise run completely independently of the activity launching them.

**Component Name**

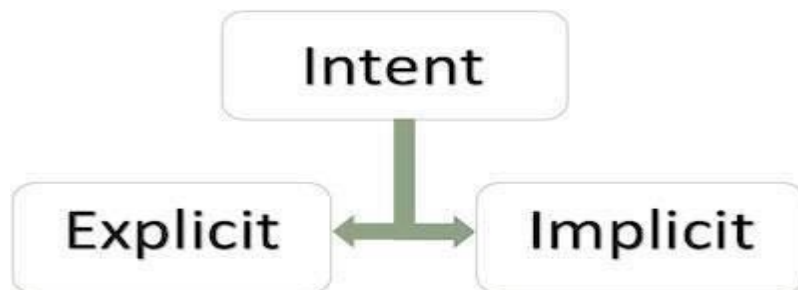
This optional field is an android **ComponentName** object representing either Activity, Service or BroadcastReceiver class. If it is set, the Intent object is delivered to an instance of the designated class otherwise Android uses other information in the Intent object to locate a suitable target.

The component name is set by `setComponent()`, `setClass()`, or `setClassName()` and read by `getComponent()`.

**Pending Intents:** A pendingIntent is a reference to a token maintained by the system. Application A can pass a pendingIntent to application B in order to allow application B to execute predefined actions on behalf of application A is still alive.

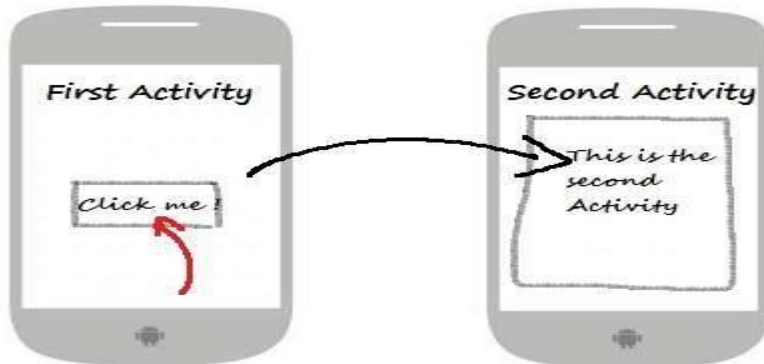
### Types of Intents

There are following two types of intents supported by Android



### Explicit Intents

Explicit intent going to be connected internal world of application, suppose if you wants to connect one activity to another activity, we can do this quote by explicit intent, below image is connecting first activity to second activity by clicking button.



These intents designate the target component by its name and they are typically used for application-internal messages - such as an activity starting a subordinate service or launching a sister activity. For example –

```
// Explicit Intent by specifying its class name
Intent i = new Intent(FirstActivity.this, SecondActivity.class);

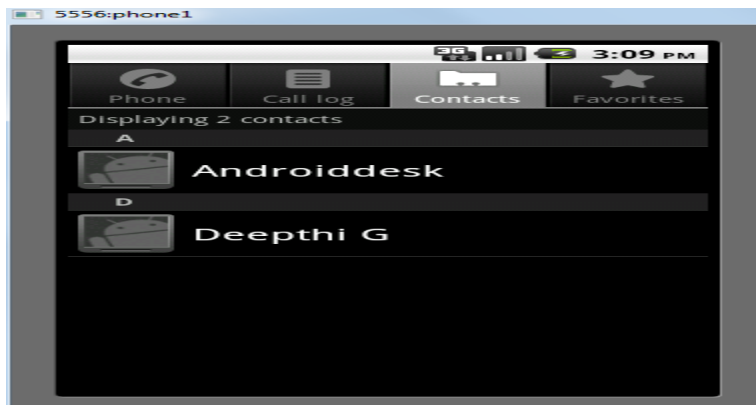
// Starts TargetActivity
startActivity(i);
```

## Implicit Intents

These intents do not name a target and the field for the component name is left blank. Implicit intents are often used to activate components in other applications. For example –

```
Intent read1=new Intent();  
read1.setAction(android.content.Intent.ACTION_VIEW);  
read1.setData(ContactsContract.Contacts.CONTENT_URI);  
startActivity(read1);
```

Above code will give result as shown below



The target component which receives the intent can use the **getExtras()** method to get the extra data sent by the source component. For example –

```
// Get bundle object at appropriate place in your code  
Bundle extras = getIntent().getExtras();  
  
// Extract data using passed keys  
String value1 = extras.getString("Key1");  
String value2 = extras.getString("Key2");
```

## Intent Filters

You have seen how an Intent has been used to call an another activity. Android OS uses filters to pinpoint the set of Activities, Services, and Broadcast receivers that can handle the Intent with help of specified set of action, categories, data scheme associated with an Intent. You will use **<intent-filter>** element in the manifest file to list down actions, categories and data types associated with any activity, service, or broadcast receiver.

Following is an example of a part of **AndroidManifest.xml** file to specify an activity **com.example.My Application.CustomActivity** which can be invoked by either of the two mentioned actions, one category, and one data –



```
<activity android:name=".CustomActivity"
    android:label="@string/app_name">

    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <action android:name="com.example.My Application.LAUNCH" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:scheme="http" />
    </intent-filter>

</activity>
```

Once this activity is defined along with above mentioned filters, other activities will be able to invoke this activity using either the **android.intent.action.VIEW**, or using the **com.example.My Application.LAUNCH** action provided their category is **android.intent.category.DEFAULT**.

The **<data>** element specifies the data type expected by the activity to be called and for above example our custom activity expects the data to start with the "http://"

There may be a situation that an intent can pass through the filters of more than one activity or service, the user may be asked which component to activate. An exception is raised if no target can be found.

There are following test Android checks before invoking an activity –

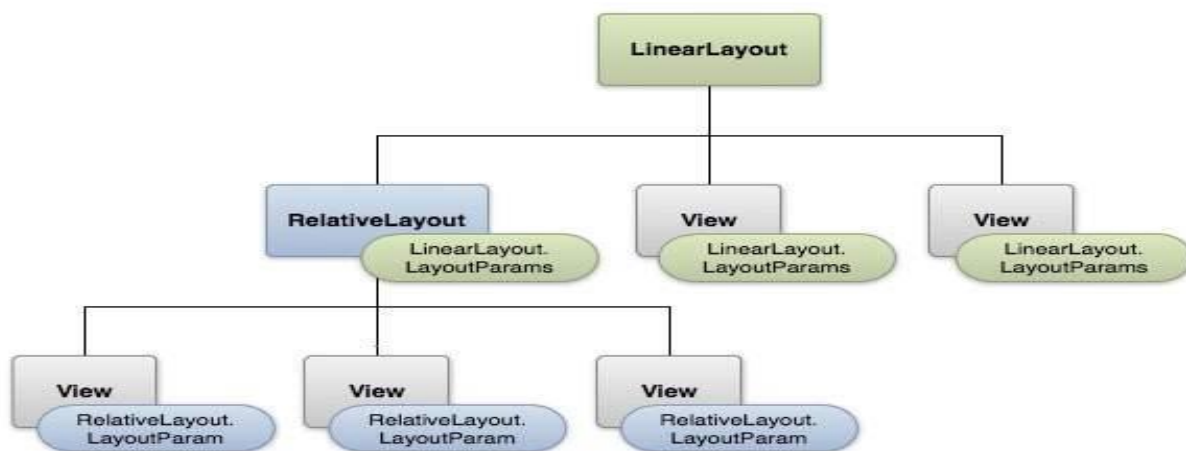
- A filter **<intent-filter>** may list more than one action as shown above, but this list cannot be empty; a filter must contain at least one **<action>** element, otherwise it will block all intents. If more than one actions are mentioned then Android tries to match one of the mentioned actions before invoking the activity.
- A filter **<intent-filter>** may list zero, one or more than one categories. if there is no category mentioned then Android always pass this test but if more than one categories are mentioned then for an intent to pass the category test, every category in the Intent object must match a category in the filter.
- Each **<data>** element can specify a URI and a data type (MIME media type). There are separate attributes like **scheme**, **host**, **port**, and **path** for each part of the URI. An Intent object that contains both a URI and a data type passes the data type part of the test only if its type matches a type listed in the filter.

## ANDROID – UI Layouts

The basic building block for user interface is a **View** object which is created from the View class and occupies a rectangular area on the screen and is responsible for drawing and event handling. View is the base class for widgets, which are used to create interactive UI components like buttons, text fields, etc.

The **ViewGroup** is a subclass of **View** and provides invisible container that hold other Views or other ViewGroups and define their layout properties.

At third level we have different layouts which are subclasses of ViewGroup class and a typical layout defines the visual structure for an Android user interface and can be created either at run time using **View/ViewGroup** objects or you can declare your layout using simple XML file **main\_layout.xml** which is located in the res/layout folder of your project.



A layout may contain any type of widgets such as buttons, labels, textboxes, and so on. Following is a simple example of XML file having LinearLayout –

```
<?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:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="This is a TextView" />

    <Button android:id="@+id/button"
        android:layout_width="wrap_content"
```

```
android:layout_height="wrap_content"  
android:text="This is a Button" />
```

```
<!-- More GUI components go here -->
```

```
</LinearLayout>
```

Once your layout has created, you can load the layout resource from your application code, in your *Activity.onCreate()* callback implementation as shown below –

```
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
}
```

### Android Layout Types

There are number of Layouts provided by Android which you will use in almost all the Android applications to provide different view, look and feel.

Sr.No	Layout & Description
1	<u>Linear Layout</u> : LinearLayout is a view group that aligns all children in a single direction, vertically or horizontally.
2	<u>Relative Layout</u> : RelativeLayout is a view group that displays child views in relative positions.
3	<u>Table Layout</u> : TableLayout is a view that groups views into rows and columns.
4	<u>Absolute Layout</u> : AbsoluteLayout enables you to specify the exact location of its children.
5	<u>Frame Layout</u> : The FrameLayout is a placeholder on screen that you can use to display a single view.
6	<u>List View</u> : ListView is a view group that displays a list of scrollable items.

7	<u>Grid View</u> : GridView is a ViewGroup that displays items in a two-dimensional, scrollable grid.
---	---

### Layout Attributes

Each layout has a set of attributes which define the visual properties of that layout. There are few common attributes among all the layouts and their are other attributes which are specific to that layout. Following are common attributes and will be applied to all the layouts:

Sr.No	Attribute & Description
1	<b>android:id</b> : This is the ID which uniquely identifies the view.
2	<b>android:layout_width</b> : This is the width of the layout.
3	<b>android:layout_height</b> : This is the height of the layout
4	<b>android:layout_marginTop</b> : This is the extra space on the top side of the layout.
5	<b>android:layout_marginBottom</b> : This is the extra space on the bottom side of the layout.
6	<b>android:layout_marginLeft</b> : This is the extra space on the left side of the layout.
7	<b>android:layout_marginRight</b> : This is the extra space on the right side of the layout.
8	<b>android:layout_gravity</b> : This specifies how child Views are positioned.
9	<b>android:layout_weight</b> : This specifies how much of the extra space in the layout should be allocated to the View.

10	<b>android:layout_x</b> : This specifies the x-coordinate of the layout.
11	<b>android:layout_y</b> : This specifies the y-coordinate of the layout.
12	<b>android:layout_width</b> : This is the width of the layout.
13	<b>android:paddingLeft</b> : This is the left padding filled for the layout.
14	<b>android:paddingRight</b> : This is the right padding filled for the layout.
15	<b>android:paddingTop</b> : This is the top padding filled for the layout.
16	<b>android:paddingBottom</b> : This is the bottom padding filled for the layout.

Here width and height are the dimension of the layout/view which can be specified in terms of dp (Density-independent Pixels), sp (Scale-independent Pixels), pt (Points which is 1/72 of an inch), px (Pixels), mm (Millimeters) and finally in (inches).

You can specify width and height with exact measurements but more often, you will use one of these constants to set the width or height –

- **android:layout\_width=wrap\_content** tells your view to size itself to the dimensions required by its content.
- **android:layout\_width=fill\_parent** tells your view to become as big as its parent view.

Gravity attribute plays important role in positioning the view object and it can take one or more (separated by '|') of the following constant values.

Constant	Value	Description
Top	0x30	Push object to the top of its container, not changing its size.
Bottom	0x50	Push object to the bottom of its container, not changing its size.

Left	0x03	Push object to the left of its container, not changing its size.
Right	0x05	Push object to the right of its container, not changing its size.
center_vertical	0x10	Place object in the vertical center of its container, not changing its size.
fill_vertical	0x70	Grow the vertical size of the object if needed so it completely fills its container.
center_horizontal	0x01	Place object in the horizontal center of its container, not changing its size.
fill_horizontal	0x07	Grow the horizontal size of the object if needed so it completely fills its container.
Center	0x11	Place the object in the center of its container in both the vertical and horizontal axis, not changing its size.
Fill	0x77	Grow the horizontal and vertical size of the object if needed so it completely fills its container.
clip_vertical	0x80	Additional option that can be set to have the top and/or bottom edges of the child clipped to its container's bounds. The clip will be based on the vertical gravity: a top gravity will clip the bottom edge, a bottom gravity will clip the top edge, and neither will clip both edges.
clip_horizontal	0x08	Additional option that can be set to have the left and/or right edges of the child clipped to its container's bounds. The clip will be based on the horizontal gravity: a left gravity will clip the right edge, a right gravity will clip the left edge, and neither will clip both edges.

Start	0x0080 0003	Push object to the beginning of its container, not changing its size.
End	0x0080 0005	Push object to the end of its container, not changing its size.

### View Identification

A view object may have a unique ID assigned to it which will identify the View uniquely within the tree. The syntax for an ID, inside an XML tag is –

`android:id="@+id/my_button"`

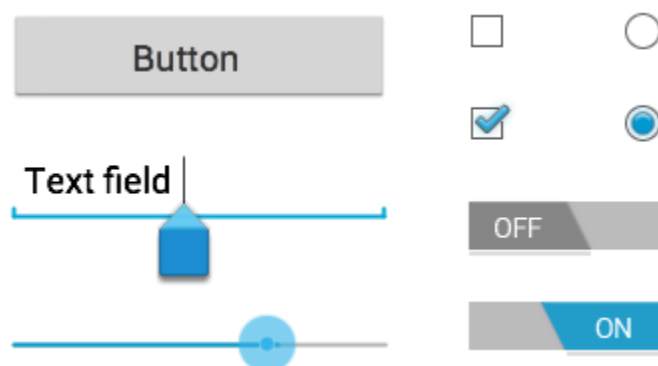
Following is a brief description of @ and + signs –

- The at-symbol (@) at the beginning of the string indicates that the XML parser should parse and expand the rest of the ID string and identify it as an ID resource.
- The plus-symbol (+) means that this is a new resource name that must be created and added to our resources. To create an instance of the view object and capture it from the layout, use the following –

```
Button myButton = (Button) findViewById(R.id.my_button);
```

### ANDROID- UI CONTROLS

Input controls are the interactive components in your app's user interface. Android provides a wide variety of controls you can use in your UI, such as buttons, text fields, seek bars, check box, zoom buttons, toggle buttons, and many more.



A **View** is an object that draws something on the screen that the user can interact with and a **ViewGroup** is an object that holds other View (and ViewGroup) objects in order to define the layout of the user interface.

You define your layout in an XML file which offers a human-readable structure for the layout, similar to HTML. For example, a simple vertical layout with a text view and a button looks like this –

```
<?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:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="I am a TextView" />

    <Button android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="I am a Button" />
</LinearLayout>
```

### Android UI Controls

There are number of UI controls provided by Android that allow you to build the graphical user interface for your app.

Sr.No.	UI Control & Description
1	<u>TextView</u> : This control is used to display text to the user.
2	<u>EditText</u> : EditText is a predefined subclass of TextView that includes rich editing capabilities.
3	<u>AutoCompleteTextView</u> : The AutoCompleteTextView is a view that is similar to EditText, except that it shows a list of completion suggestions automatically while the user is typing.



4	<u>Button</u> : A push-button that can be pressed, or clicked, by the user to perform an action.
5	<u>ImageButton</u> : An ImageButton is an AbsoluteLayout which enables you to specify the exact location of its children. This shows a button with an image (instead of text) that can be pressed or clicked by the user.
6	<u>CheckBox</u> : An on/off switch that can be toggled by the user. You should use check box when presenting users with a group of selectable options that are not mutually exclusive.
7	<u>ToggleButton</u> : An on/off button with a light indicator.
8	<u>RadioButton</u> : The RadioButton has two states: either checked or unchecked.
9	<u>RadioGroup</u> : A RadioGroup is used to group together one or more RadioButtons.
10	<u>ProgressBar</u> : The ProgressBar view provides visual feedback about some ongoing tasks, such as when you are performing a task in the background.
11	<u>Spinner</u> : A drop-down list that allows users to select one value from a set.
12	<u>TimePicker</u> : The TimePicker view enables users to select a time of the day, in either 24-hour mode or AM/PM mode.
13	<u>DatePicker</u> : The DatePicker view enables users to select a date of the day.

### Create UI Controls

Input controls are the interactive components in your app's user interface. Android provides a wide variety of controls you can use in your UI, such as buttons, text fields, seek bars, check box, zoom buttons, toggle buttons, and many more.

As explained in previous chapter, a view object may have a unique ID assigned to it which will identify the View uniquely within the tree. The syntax for an ID, inside an XML tag is –

```
android:id="@+id/text_id"
```

To create a UI Control/View/Widget you will have to define a view/widget in the layout file and assign it a unique ID as follows –

```
<?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:id="@+id/text_id"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="I am a TextView" />
</LinearLayout>
```

Then finally create an instance of the Control object and capture it from the layout, use the following –

```
TextView myText = (TextView) findViewById(R.id.text_id);
```

## EVENT HANDLING

**Events are a useful way to collect data about a user's interaction with interactive components of Applications.** Like button presses or screen touch etc. The Android framework maintains an event queue as first-in, first-out (FIFO) basis. You can capture these events in your program and take appropriate action as per requirements.

There are following three concepts related to Android Event Management –

- **Event Listeners** – An event listener is an interface in the View class that contains a single callback method. These methods will be called by the Android framework when the View to which the listener has been registered is triggered by user interaction with the item in the UI.
- **Event Listeners Registration** – Event Registration is the process by which an Event Handler gets registered with an Event Listener so that the handler is called when the Event Listener fires the event.
- **Event Handlers** – When an event happens and we have registered an event listener for the event, the event listener calls the Event Handlers, which is the method that actually handles the event.

**List of Event Listeners & Event Handlers**

Event Handler	Event Listener & Description
onClick()	<b>OnClickListener()</b> This is called when the user either clicks or touches or focuses upon any widget like button, text, image etc. You will use onClick() event handler to handle such event.
onLongClick()	<b>OnLongClickListener()</b> This is called when the user either clicks or touches or focuses upon any widget like button, text, image etc. for one or more seconds. You will use onLongClick() event handler to handle such event.
onFocusChange()	<b>OnFocusChangeListener()</b> This is called when the widget loses its focus i.e. user goes away from the view item. You will use onFocusChange() event handler to handle such event.
onKey()	<b>OnFocusChangeListener()</b> This is called when the user is focused on the item and presses or releases a hardware key on the device. You will use onKey() event handler to handle such event.
onTouch()	<b>OnTouchListener()</b> This is called when the user presses the key, releases the key, or any movement gesture on the screen. You will use onTouch() event handler to handle such event.

onMenuItemClick()	<b>OnMenuItemClickListener()</b> This is called when the user selects a menu item. You will use onMenuItemClick() event handler to handle such event.
onCreateContextMenu()	<b>onCreateContextMenuListener()</b> This is called when the context menu is being built(as the result of a sustained "long click")

There are many more event listeners available as a part of **View** class like OnHoverListener, OnDragListener etc which may be needed for your application. So I recommend to refer official documentation for Android application development in case you are going to develop a sophisticated apps.

### Event Listeners Registration

Event Registration is the process by which an Event Handler gets registered with an Event Listener so that the handler is called when the Event Listener fires the event. Though there are several tricky ways to register your event listener for any event, but I'm going to list down only top 3 ways, out of which you can use any of them based on the situation.

- Using an Anonymous Inner Class
- Activity class implements the Listener interface.
- Using Layout file activity\_main.xml to specify event handler directly.

Below section will provide you detailed examples on all the three scenarios –

### Touch Mode

Users can interact with their devices by using hardware keys or buttons or touching the screen. Touching the screen puts the device into touch mode. The user can then interact with it by touching the on-screen virtual buttons, images, etc. You can check if the device is in touch mode by calling the View class's `isInTouchMode()` method.

### Focus

A view or widget is usually highlighted or displays a flashing cursor when it's in focus. This indicates that it's ready to accept input from the user.

- **isFocusable()** – it returns true or false

- **isFocusableInTouchMode()** – checks to see if the view is focusable in touch mode. (A view may be focusable when using a hardware key but not when the device is in touch mode)

```
android:focusUp="@=id/button_1"
```

### onTouchEvent()

```
public boolean onTouchEvent(motionEvent event){
    switch(event.getAction()){
        case TOUCH_DOWN:
            Toast.makeText(this,"you have clicked down Touch button",Toast.LENGTH_LONG).show();
            break();

        case TOUCH_UP:
            Toast.makeText(this,"you have clicked up touch button",Toast.LENGTH_LONG).show();
            break;

        case TOUCH_MOVE:
            Toast.makeText(this,"you have clicked move touch button"Toast.LENGTH_LONG).show();
            break;
    }
    return super.onTouchEvent(event) ;
}
```

## STYLES AND THEMES

A **style** resource defines the format and look for a UI. A style can be applied to an individual View (from within a layout file) or to an entire Activity or application (from within the manifest file).

### Defining Styles

A style is defined in an XML resource that is separate from the XML that specifies the layout. This XML file resides under **res/values/** directory of your project and will have **<resources>** as the root node which is mandatory for the style file. The name of the XML file is arbitrary, but it must use the .xml extension.

You can define multiple styles per file using **<style>** tag but each style will have its name that uniquely identifies the style. Android style attributes are set using **<item>** tag as shown below –

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

```
<resources>
  <style name="CustomFontStyle">
    <item name="android:layout_width">fill_parent</item>
    <item name="android:layout_height">wrap_content</item>
    <item name="android:capitalize">characters</item>
    <item name="android:typeface">monospace</item>
    <item name="android:textSize">12pt</item>
    <item name="android:textColor">#00FF00</item>/>
  </style>
</resources>
```

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.

**Using Styles:** Once your style is defined, you can use it in your XML Layout file using style attribute as follows –

```
<?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:id="@+id/text_id"
    style="@style/CustomFontStyle"
    android:text="@string/hello_world" />

</LinearLayout>.
```

## Style Inheritance

Android supports style Inheritance in very much similar way as cascading style sheet in web design. You can use this to inherit properties from an existing style and then define only the properties that you want to change or add.

To implement a custom theme create or edit MyAndroidApp/res/values/themes.xml and add the following –

```
<resources>
...
  <style name="MyCustomTheme" parent="android:style/Theme">
    <item name="android:textColorPrimary">#ffff0000</item>
```

```
</style>
...
</resources>
```

In your AndroidManifest.xml apply the theme to the activities you want to style –

```
<activity
    android:name="com.myapp.MyActivity"
    ...
    android:theme="@style/MyCustomTheme"
/>
```

Your new theme will be applied to your activity, and text is now bright red.



### Applying Colors to Theme Attributes

Your color resource can then be applied to some theme attributes, such as the window background and the primary text color, by adding <item> elements to your custom theme. These attributes are defined in your styles.xml file. For example, to apply the custom color to the window background, add the following two <item> elements to your custom theme, defined in MyAndroidApp/res/values/styles.xml file –

```
<resources>
...
<style name="MyCustomTheme" ...>
    <item name="android:windowBackground">@color/my_custom_color</item>
    <item name="android:colorBackgroundCacheHint">@color/my_custom_color</item>
</style>
...
</resources>
```



### Using a Custom Nine-Patch With Buttons

A nine-patch drawable is a special kind of image which can be scaled in width and height while maintaining its visual integrity. Nine-patches are the most common way to specify the appearance of Android buttons, though any drawable type can be used.



*a Sample of Nine-Patch button*

### Steps to create Nine-Patch Buttons

- Save this bitmap as /res/drawable/my\_nine\_patch.9.png
- Define a new style
- Apply the new button style to the buttonStyle attribute of your custom theme

### Define a new Style

```
<resources>
...
<style name="MyCustomButton" parent="android:Widget.Button">
  <item name="android:background">@drawable/my_nine_patch</item>
</style>
...
</resources>
```

### Apply the theme

```
<resources>
...
<style name="MyCustomTheme" parent="...">
  ...
  <item name="android:buttonStyle">@style/MyCustomButton</item>
</style>
```



```
...  
</resources>
```



## Android Themes

Hope you understood the concept of Style, so now let's try to understand what is a **Theme**. A theme is nothing but an Android style applied to an entire Activity or application, rather than an individual View.

Thus, when a style is applied as a theme, every **View** in the Activity or application will apply each style property that it supports. For example, you can apply the same **CustomFontStyle** style as a theme for an Activity and then all text inside that **Activity** will have green monospace font.

To set a theme for all the activities of your application, open the **AndroidManifest.xml** file and edit the **<application>** tag to include the **android:theme** attribute with the style name. For example –

```
<application android:theme="@style/CustomFontStyle">
```

But if you want a theme applied to just one Activity in your application, then add the **android:theme** attribute to the **<activity>** tag only. For example –

```
<activity android:theme="@style/CustomFontStyle">
```

There are number of default themes defined by Android which you can use directly or inherit them using **parent** attribute as follows –

```
<style name="CustomTheme" parent="android:Theme.Light">  
...  
</style>
```

To understand the concept related to Android Theme, you can check Theme Demo Example.

### Styling the color palette

The layout design can implementable based on them based colours, for example as following design is designed based on them colour(blue)



Above layout has designed based on style.xml file, Which has placed at **res/values/**

```
<resource>
  <style name="AppTheme" parent="android:Theme.Material">
    <item name="android:color/primary">@color/primary</item>
    <item name="android:color/primaryDark">@color/primary_dark</item>
    <item name="android:colorAccent/primary">@color/accent</item>
  </style>
</resource>
```

## Default Styles & Themes

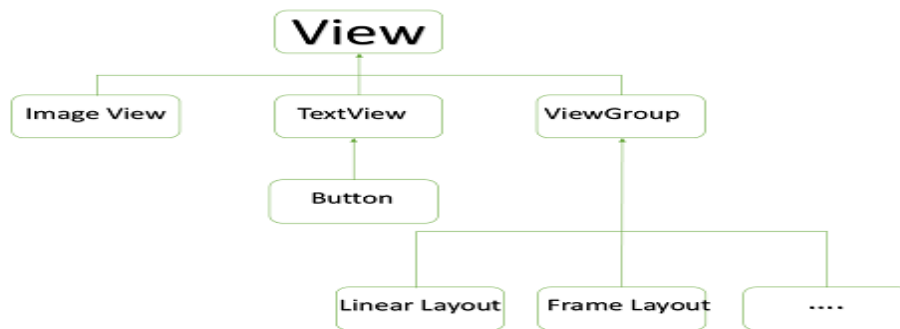
The Android platform provides a large collection of styles and themes that you can use in your applications. You can find a reference of all available styles in the **R.style** class. To use the styles listed here, replace all underscores in the style name with a period. For example, you can apply the Theme\_NoTitleBar theme with "@android:style/Theme.NoTitleBar". You can see the following source code for Android styles and themes –

- Android Styles (styles.xml)
- Android Themes (themes.xml)

## Android – Custom Components

Android offers a great list of pre-built widgets like Button, TextView, EditText, ListView, CheckBox, RadioButton, Gallery, Spinner, AutoCompleteTextView etc. which you can use directly in your Android application development, but there may be a situation when you are not satisfied with existing functionality of any of the available widgets. Android provides you with means of creating your own custom components which you can customized to suit your needs.

If you only need to make small adjustments to an existing widget or layout, you can simply subclass the widget or layout and override its methods which will give you precise control over the appearance and function of a screen element.



## Android - UI Design

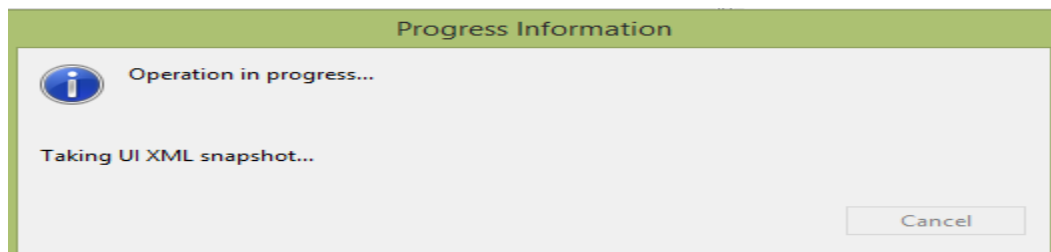
In this chapter we will look at the different UI components of android screen. This chapter also covers the tips to make a better UI design and also explains how to design a UI.

### UI screen components

A typical user interface of an android application consists of action bar and the application content area.

- Main Action Bar
- View Control
- Content Area
- Split Action Bar

These components have also been shown in the image below –



### Understanding Screen Components

The basic unit of android application is the activity. A UI is defined in an xml file. During compilation, each element in the XML is compiled into equivalent Android GUI class with attributes represented by methods.

### View and ViewGroups

An activity is consist of views. A view is just a widget that appears on the screen. It could be button e.t.c. One or more views can be grouped together into one GroupView. Example of ViewGroup includes layouts.

### Types of Android view:

- TextView
- EditText
- Button
- ImageButton
- Date Picker
- RadioButton
- CheckBox Button
- ImageView

### Types of layout

There are many types of layout. Some of which are listed below –

- Linear Layout
- Absolute Layout
- Table Layout
- Frame Layout
- Relative Layout

### Linear Layout

Linear layout is further divided into horizontal and vertical layout. It means it can arrange views in a single column or in a single row. Here is the code of linear layout(vertical) that includes a text view.

```
<?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>
```

### Absolute Layout

The AbsoluteLayout enables you to specify the exact location of its children. It can be declared like this.

```
<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>
```

### Table Layout

The TableLayout groups views into rows and columns. It can be declared like this.

```
<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>
```

## Relative Layout

The Relative Layout enables you to specify how child views are positioned relative to each other. It can be declared like this.

```
<RelativeLayout
    android:id="@+id/RLLayout"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    xmlns:android="http://schemas.android.com/apk/res/android" >
</RelativeLayout>
```

## FrameLayout

The FrameLayout is a placeholder on screen that you can use to display a single view. It can be declared like this.

```
<?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>
```

Apart from these attributes, there are other attributes that are common in all views and ViewGroups. They are listed below –

Sr.No	View & description
1	<b>layout_width</b> : Specifies the width of the View or ViewGroup
2	<b>layout_height</b> : Specifies the height of the View or ViewGroup

3	<b>layout_marginTop:</b> Specifies extra space on the top side of the View or ViewGroup
4	<b>layout_marginBottom:</b> Specifies extra space on the bottom side of the View or ViewGroup
5	<b>layout_marginLeft:</b> Specifies extra space on the left side of the View or ViewGroup
6	<b>layout_marginRight:</b> Specifies extra space on the right side of the View or ViewGroup
7	<b>layout_gravity:</b> Specifies how child Views are positioned
8	<b>layout_weight:</b> Specifies how much of the extra space in the layout should be allocated to the View

### Units of Measurement

When you are specifying the size of an element on an Android UI, you should remember the following units of measurement.

Sr.No	Unit & description
1	<b>dp: Density-independent pixel.</b> 1 dp is equivalent to one pixel on a 160 dpi screen.
2	<b>sp: Scale-independent pixel.</b> This is similar to dp and is recommended for specifying font sizes
3	<b>pt: Point.</b> A point is defined to be 1/72 of an inch, based on the physical screen size.
4	<b>px :Pixel.</b> Corresponds to actual pixels on the screen

### Screen Densities

Sr.No	Density & DPI
1	<b>Low density (ldpi):</b> 120 dpi
2	<b>Medium density (mdpi):</b> 160 dpi
3	<b>High density (hdpi):</b> 240 dpi
4	<b>Extra High density (xhdpi):</b> 320 dpi

### Optimizing layouts

Here are some of the guidelines for creating efficient layouts.

- Avoid unnecessary nesting
- Avoid using too many Views
- Avoid deep nesting

## Android - UI Patterns

### UI Patterns components

A good android application should follow following UI patterns –

- Action Bar
- Confirming and Acknowledging
- Settings
- Help
- Selection

Now we will discuss the above mentioned UI Patterns in detail.

### Action Bar



The action bar is a dedicated bar at the top of each screen that is generally persistent throughout the app. It provides you several key function which are as following –

- Makes important actions prominent and accessible
- Supports consistent navigation and view switching within apps
- Reduces clutter by providing an action overflow for rarely used actions
- Provides a dedicated space for giving your app an identity

### Action Bar Components

Action Bar has four major components which can be seen in the following image.



These components name and functionality is discussed below –

Sr.No	Action Bar Components
1	<b>App Icon:</b> The app icon establishes your app's identity. It can be replaced with a different logo or branding if you wish.
2	<b>View control:</b> If your app displays data in different views, this segment of the action bar allows users to switch views.
3	<b>Action buttons:</b> Show the most important actions of your app in the actions section.
4	<b>Action overflow:</b> Move less often used actions to the action overflow.

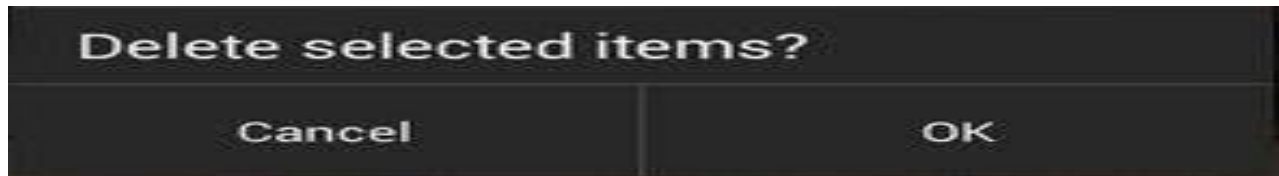
### Confirming and Acknowledging

When a user invokes a action on your app's UI, it is a good practice to **confirm** or **acknowledge** that action through a toast or a dialog box.

There is a difference between Confirming and Acknowledging.

### Confirming

When we ask the user to verify that they truly want to proceed with a action that they just invoked, it is called confirming. As you can see in the following image –



### Acknowledging

When we display a toast to let the user know that the action they just invoked has been completed, this is called acknowledging, As you can see in the following image –



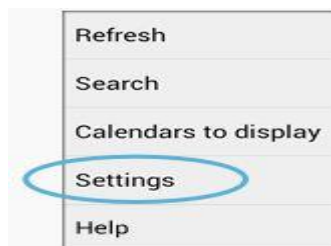
### Settings

The place in your app where users can indicate their preferences for how your app should behave is called as Settings. The use of settings can benefit your app's users in the following ways –

- **Settings** help user to predetermine that what will happen in certain situations
- Use of **settings** in your app help users to feel in control

### Placement of Settings

It is preferred by the android developers to always make "settings" option part of action overflow which is mentioned above. As users did not frequently use this option so the common practice is to place it below all other items except "Help". As you can see in the following picture –



### Help

Some of your app users may run into some difficulty while using your app and they will be looking for some answers, and they want them within the app. So always make "help" part of your app.

### Placement of Help

Like "Settings" the standard design of placing "Help" option is in **action overflow**. Always make it very last item in the menu and always label it "Help". Even if your app screen has no other action overflow items, "Help" should appear there. As you can see this in the following picture –



### Selection

Android 3.0 version changed the long press gesture to the global gesture to select data. The long press gesture is now used to select data, combining contextual actions and selection management functions for selected data into a new element called the **contextual action bar (CAB)**.

### Using Contextual Action Bar (CAB)

The selection CAB is a temporary action bar that overlays your app's current action bar while data is selected. It appears after the user long presses on a selectable data item. As you can see in the following picture –



From the CAB bar user can perform following actions –

- Select additional data items by touching them
- Trigger an action from the CAB that applies to all highlighted data items
- Dismiss the CAB via the navigation bar's Back button or the CAB's checkmark button

## Android - UI Testing

Android SDK provides the following tools to support automated, functional UI testing on your application.

- uiautomatorviewer
- uiautomator

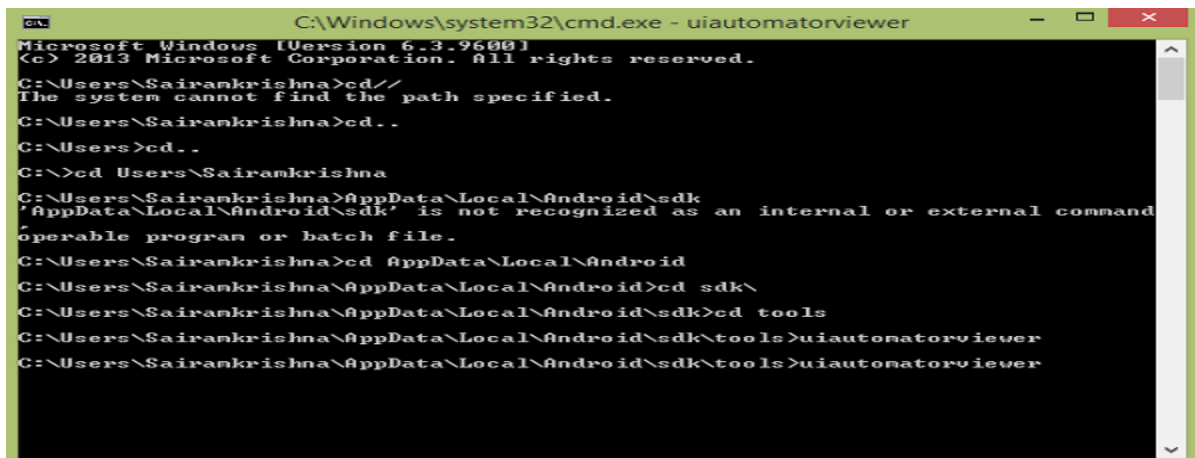
**uiautomatorviewer:** A GUI tool to scan and analyze the UI components of an Android application. The uiautomatorviewer tool provides a convenient visual interface to inspect the layout hierarchy and view the properties of the individual UI components that are displayed on the test device. Using this information, you can later create uiautomator tests with selector objects that target specific UI components to test.

To analyse the UI components of the application that you want to test, perform the following steps after installing the application given in the example.

- Connect your Android device to your development machine
- Open a terminal window and navigate to <android-sdk>/tools/
- Run the tool with this command

**uiautomatorviewer**

Commands would be followed as shown below



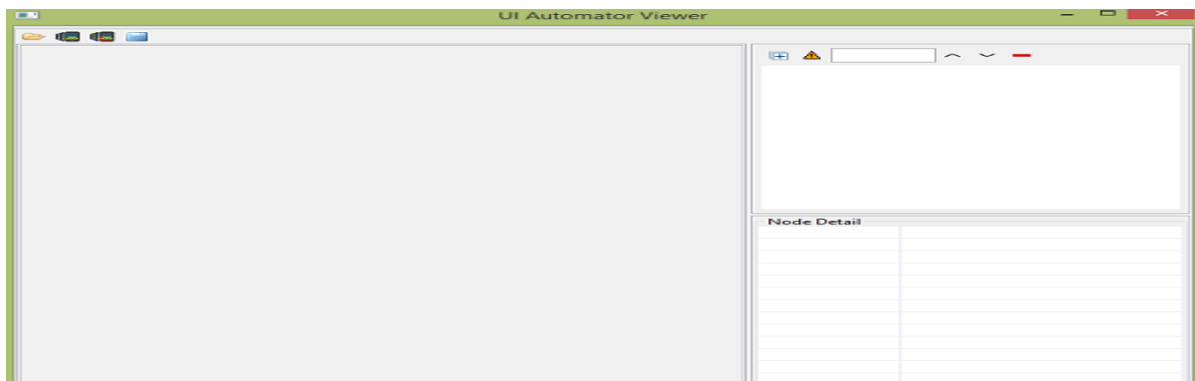
```
C:\Windows\system32\cmd.exe - uiautomatorviewer
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\Sairankrishna>cd //
The system cannot find the path specified.

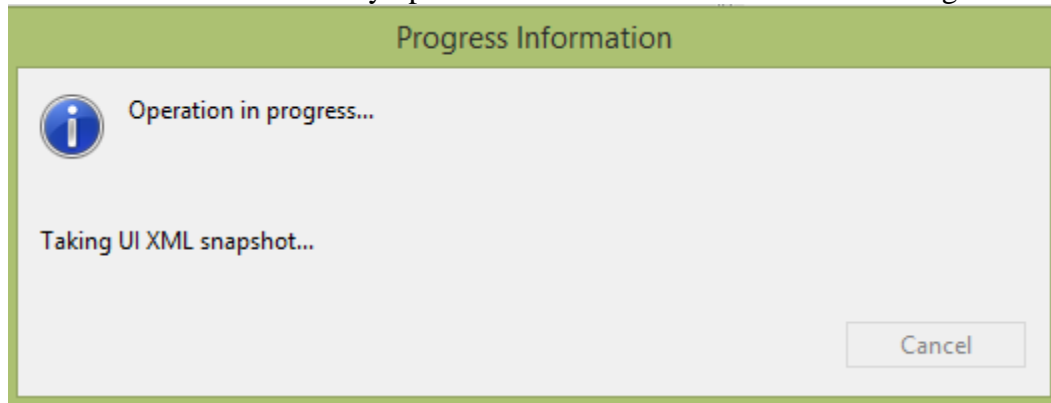
C:\Users\Sairankrishna>cd ..
C:\Users>cd ..
C:\>cd Users\Sairankrishna
C:\Users\Sairankrishna>AppData\Local\Android\sdk
'AppData\Local\Android\sdk' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\Sairankrishna>cd AppData\Local\Android
C:\Users\Sairankrishna\AppData\Local\Android>cd sdk\
C:\Users\Sairankrishna\AppData\Local\Android\sdk>cd tools
C:\Users\Sairankrishna\AppData\Local\Android\sdk\tools>uiautomatorviewer
C:\Users\Sairankrishna\AppData\Local\Android\sdk\tools>uiautomatorviewer
```

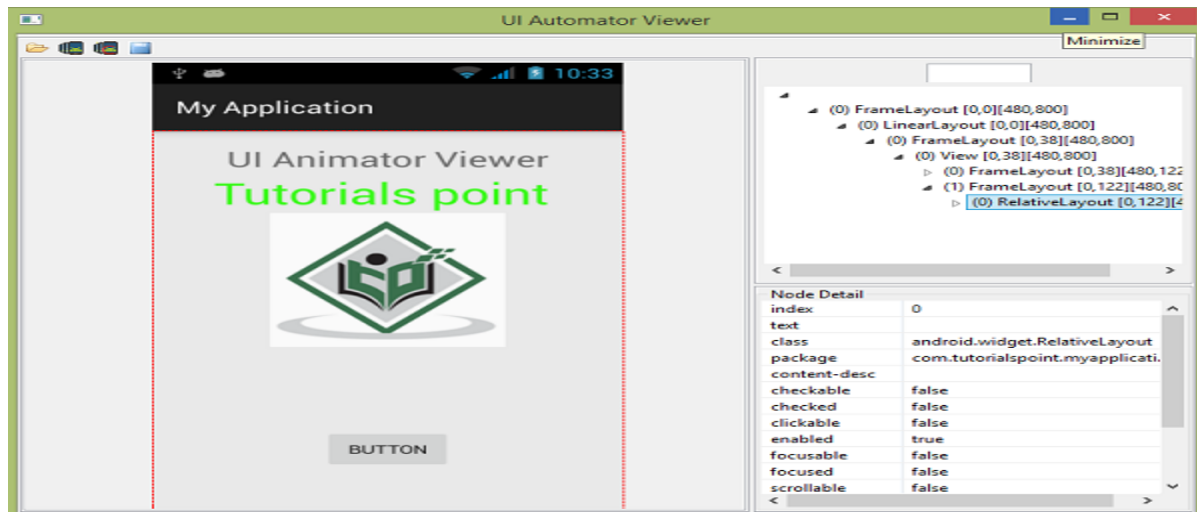
You will see the following window appear. It is the default window of the UI Automator Viewer.



- Click on the devices icon at the top right corner. It will start taking the UI XML snapshot of the screen currently opened in the device. It would be something like this.

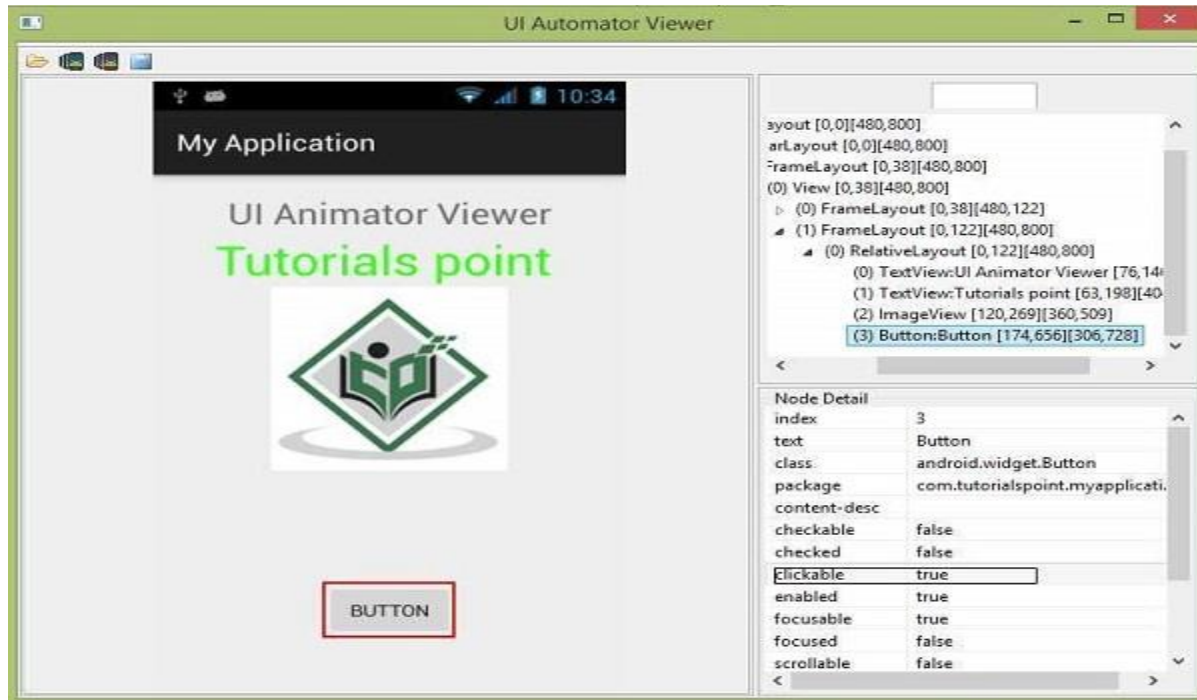


After that, you will see the snapshot of your device screen in the uiautomatorviewer window.

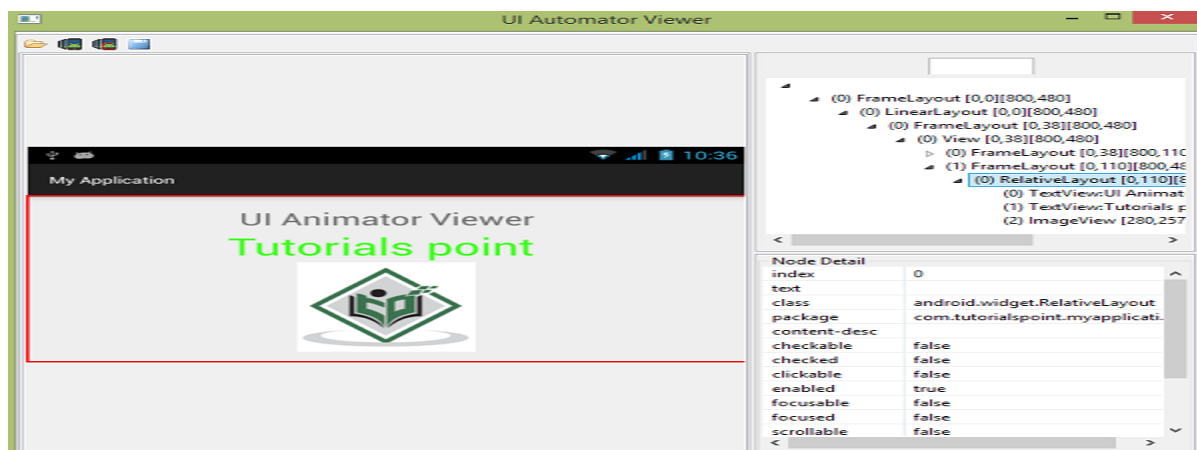


On the right side of this window, you will see two partitions. The upper partition explains the Nodes structure, the way the UI components are arranged and contained. Clicking on each node gives detail in the lower partition.

As an example, consider the below figure. When you click on the button, you can see in the upper partition that Button is selected, and in the lower partition, its details are shown. Since this button is click able, that's why its property of click able is set to true.



UI Automator Viewer also helps you to examine your UI in different orientations. For example, just change your device orientation to landscape, and again capture the screen shot. It is shown in the figure below –



## UI automator

Now you can create your own test cases and run it with uiautomatorviewer to examine them. In order to create your own test case, you need to perform the following steps –

- From the Project Explorer, right-click on the new project that you created, then select Properties > Java Build Path, and do the following –

RAKSHITHA H J- ASSISTANT PROFESSOR

DEPARTMENT OF B.Voc IN SOFTWARE AND APPLICATION DEVELOPMENT

SDMC-UJIRE

- Click Add Library > JUnit then select JUnit3 to add JUnit support.
- Click Add External JARs... and navigate to the SDK directory. Under the platforms directory, select the latest SDK version and add both the uiautomator.jar and android.jar files.
- Extend your class with UiAutomatorTestCase
- Right the necessary test cases.
- Once you have coded your test, follow these steps to build and deploy your test JAR to your target Android test device.
- Create the required build configuration files to build the output JAR. To generate the build configuration files, open a terminal and run the following command:  
`<android-sdk>/tools/android create uitest-project -n <name> -t 1 -p <path>`
- The <name> is the name of the project that contains your uiautomator test source files, and the <path> is the path to the corresponding project directory.
- From the command line, set the ANDROID\_HOME variable.  
`set ANDROID_HOME=<path_to_your_sdk>`
- Go to the project directory where your build.xml file is located and build your test JAR.  
`ant build`
- Deploy your generated test JAR file to the test device by using the adb push command.  
`adb push <path_to_output_jar> /data/local/tmp/`
- Run your test by following command –  
`adb shell uiautomator runtest LaunchSettings.jar -c com.uia.example.my.LaunchSettings`

**Dalvik Debug Monitor Service:** Android ships with a debugging tool called Dalvik Debug Monitor Server (DDMS), which provide port-forwarding services, screen capture on the device, thread and heap information on the device, logcat, process, and radio state information, incoming call and SMS spoofing, location data spoofing and more.

## What is Toast? Explain How to customize it?

Toast is used to display information for a period of time. It contains a message to be displayed quickly and disappears after specified period of time. It does not block the user interaction. Toast is a subclass of Object class. In this we use two constants for setting the duration for the Toast. Toast notification in android always appears near the bottom of the screen. We can also create our custom toast by using custom layout(xml file).

### *Steps for Implementation of Custom Toast In Android:*

**Step 1:** Firstly Retrieve the Layout Inflater with `getLayoutInflater()` (or `getSystemService()`) and then inflate the layout from XML using `inflate(int, ViewGroup)`. In inflate method first parameter is the layout resource ID and the second is the root View.

**Step 2:** Create a new Toast with Toast(Context) and set some properties of the Toast, such as the duration and gravity.

**Step 3:** Call setContentView(View) and pass the inflated layout in this method.

**Step 4:** Display the Toast on the screen using show() method of Toast.

**Example demonstrate about how to create custom toast message in android.**

**Step 1** – Create a new project in Android Studio, go to File ⇒ New Project and fill all required details to create a new project.

**Step 2** – Add the following code to res/layout/activity\_main.xml.

```
<?xml version = "1.0" encoding = "utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android = "http://schemas.android.com/apk/res/android" xmlns:tools
    = "http://schemas.android.com/tools" android:layout_width = "match_parent"
    android:layout_height = "match_parent">
    <LinearLayout
        android:layout_width = "match_parent"
        android:layout_height = "match_parent"
        android:background = "#797979"
        android:gravity = "center"
        android:orientation = "vertical">
        <Button
            android:id = "@+id/showToast"
            android:text = "Show Toast"
            android:layout_width = "wrap_content"
            android:layout_height = "wrap_content" />
        </LinearLayout>
    </android.support.constraint.ConstraintLayout>
```

**Step 3** – Add the following code to src/MainActivity.java

```
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.view.LayoutInflater;
```



```
import android.view.View;
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.Toast;
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button button = findViewById(R.id.showToast);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                LayoutInflater li = getLayoutInflater();
                View layout = li.inflate(R.layout.custom_toast,
                (ViewGroup) findViewById(R.id.custom_toast_layout_id));
                Toast toast = new Toast(getApplicationContext());
                toast.setDuration(Toast.LENGTH_SHORT);
                toast.setView(layout); //setting the view of custom toast layout
                toast.show();
            }
        });
    }
}
```

**Step 4** – Now create custom toast layout in res/layout/ custom\_toast.xml and add the following code


```
<?xml version = "1.0" encoding = "utf-8"?>
<LinearLayout xmlns:android = "http://schemas.android.com/apk/res/android"
    android:id = "@+id/custom_toast_layout_id"
    android:layout_width = "match_parent"
```

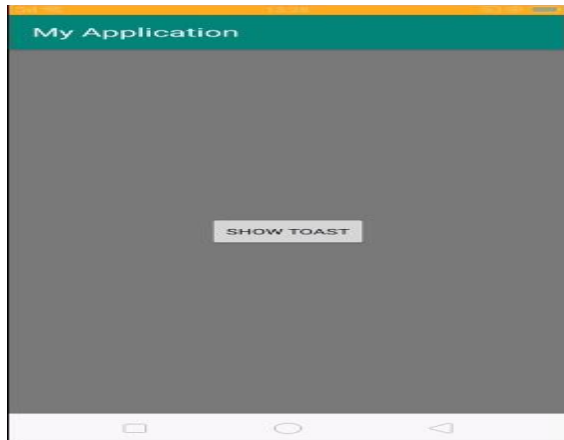
```
android:layout_height = "match_parent"
android:gravity = "center">
<LinearLayout
    android:layout_width = "wrap_content"
    android:layout_height = "62dp"
    android:gravity = "center"
    android:background = "@drawable/buttonshape"
    android:orientation = "horizontal">
    <ImageView
        android:id = "@+id/imageView"
        android:layout_width = "100dp"
        android:layout_height = "50dp"
        android:scaleType = "fitStart"
        android:src = "@drawable/logo" />
    <TextView
        android:id = "@id/text"
        android:layout_width = "wrap_content"
        android:layout_height = "wrap_content"
        android:layout_marginLeft = "10dp"
        android:layout_marginRight = "20dp"
        android:text = "This is custom toast"
        android:textColor = "#FFF"
        android:textSize = "15sp"
        android:textStyle = "bold" />
    </LinearLayout>
</LinearLayout>
```

**Step 5** – In the above code we added background for layout as buttonshape in drawable so create a xml file in drawable as buttonshape.xml and add the following code

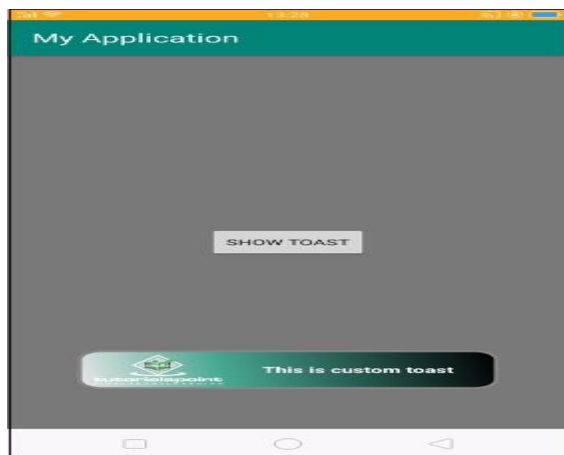
```
<?xml version = "1.0" encoding = "utf-8"?>
```

```
<shape xmlns:android = "http://schemas.android.com/apk/res/android" android:shape =  
"rectangle" >  
    <corners  
        android:radius = "14dp"  
    />  
    <gradient  
        android:angle = "45"  
        android:centerX = "%"  
        android:centerColor = "#47A891"  
        android:startColor = "#E8E8E8"  
        android:endColor = "#000000"  
        android:type = "linear"/>  
    <padding  
        android:left = "0dp"  
        android:top = "0dp"  
        android:right = "0dp"  
        android:bottom = "0dp"/>  
    <size  
        android:width = "270dp"  
        android:height = "60dp"/>  
    <stroke  
        android:width = "3dp"  
        android:color = "#878787"/>  
</shape>
```

Let's try to run your application. I assume you have connected your actual Android Mobile device with your computer. To run the app from android studio, open one of your project's activity files and click Run  icon from the toolbar. Select your mobile device as an option and then check your mobile device which will display your default screen



Now click on Show Toast button, it will give custom toast result as shown below



## Android Debug Bridge (adb)

Android Debug Bridge (adb) is a versatile command-line tool that lets you communicate with a device. The ADB is typically used to communicate with a smartphone, tablet, smartwatch, set-top box, or any other device that can run the Android operating system. We can do things on an Android device that may not be suitable for everyday use, like, install apps outside of the Play Store, debug apps, access hidden features, and bring up a UNIX shell, etc. For security reasons, Developer Options need to be unlocked and you need to have USB Debugging Mode enabled as well. Not only that, but you also need to authorize USB Debugging access to the specific PC that you're connected to with a USB cable. It is a client-server program that includes three components –

- **A client**, which sends commands. The client runs on your development machine. You can invoke a client from a command-line terminal by issuing an adb command.

- A **daemon**, which runs commands on a device. The daemon runs as a background process on each device.
- A **server**, which manages communication between the client and the daemon. The server runs as a background process on your development machine.

### How it works

After starting an **adb** client in the kali Linux terminal, the client first confirms whether there is an **adb** server process already running. If there isn't, it starts the server process. When the server starts, it binds to local TCP port 5037 and listens for commands sent from adb clients—all adb clients use port 5037 to communicate with the **adb** server. The server then sets up connections to all running devices. It locates emulators by scanning odd-numbered ports in the range 5555 to 5585, the range used by the first 16 emulators. Where the server finds an **adb** daemon, it sets up a connection to that port.

**Note** – To use adb with a device connected over USB, you must enable **USB debugging** in the device system settings, under **Developer options**. The Developer options screen is hidden by default. To make it visible, go to **Settings > About phone** and tap **Build number** seven times. Return to the previous screen to find **Developer options** at the bottom.

### Connection

- After enabling the developer option in the android device, connect it to the PC with the USB cable. However, we can make a connection through Wi-Fi too.
- Open the terminal in the Kali Linux
- Connect to the device by its IP address which is found at **Settings > About tablet (or About phone) > Status > IP address**  
:/> adb connect 'ip address'
- After that, confirm that your host computer is connected to the target device by the following command; it shows the serial number of the connected devices.  
:/> adb devices

### ADB Commands

The user can perform multiple types of operator once a connection is established to the android device. Here, the list of commands to communicate with device as following

Command	Comments
adb devices	Print the connected devices

Command	Comments
Adb kill-server	Kill the adb Server
adb root	To get root access
adb wait-for-devices	Wait for adb devices
adb shell stop thermal-engine	Stopping system service /system/bin/thermal-engine
Adb install	Install an application in adb
adb shell	Initiate a shell
Sadb shell dumpsys	Shows memory consumption details
adb shell echo performance > /sys/devices/system/cpu/cpu1/cpufreq/scaling_governor	Put the cpu1 scaling governor to performance mode
Adb pull	Copy a file or directory from the device
Adb push	Copy a file or directory to the device
Adb forwarding tcp:6100:7100	Port forwarding