

Android is a mobile operating system (OS) currently developed by **Google**, based on the **Linux kernel** and designed primarily for touch screen mobile devices such as smart phones and tablets. Android's user interface(UI) is mainly based on direct manipulation, using touch gestures that loosely correspond to real-world actions, such as swiping, tapping and pinching, to manipulate on-screen objects, along with a virtual keyboard for text input. Android programming is based on Java programming language so if you have basic understanding on Java programming then it will be a fun to learn Android application development.

What is Android?

Android is an open source and Linux-based **Operating System** for mobile devices such as smart phones and tablet computers. Android was developed by the *Open Handset Alliance*, led by Google, and other companies.

The first beta version of the Android Software Development Kit (SDK) was released by Google in 2007 where as the first commercial version, Android 1.0, was released in September 2008. On June 27, 2012, at the Google I/O conference, Google announced the next Android version, 4.1 **Jelly Bean**. Jelly Bean is an incremental update, with the primary aim of improving the user interface, both in terms of functionality and performance.

The source code for Android is available under free and open source software licenses. Google publishes most of the code under the Apache License version 2.0 and the rest, Linux kernel changes, under the GNU General Public License version 2.

Features of Android

Feature	Description
Beautiful UI	Android OS basic screen provides a beautiful and intuitive user interface.
Connectivity	GSM/EDGE, CDMA, UMTS, Bluetooth, Wi-Fi, LTE, NFC and WiMAX.
Storage	SQLite is a lightweight relational database, used for data storage purposes.
Media support	H.263, H.264, MPEG-4 SP, AMR, AMR-WB, AAC, HE-AAC, AAC 5.1, MP3, MIDI, Ogg Vorbis, WAV, JPEG, PNG, GIF, and BMP
Messaging	SMS and MMS
Web browser	Based on the open-source WebKit layout engine, coupled with Chrome's V8 JavaScript engine supporting HTML5 and CSS3.
Multi-touch	Android has native support for multi-touch which was initially made available in handsets such as the HTC Hero.
Multi-tasking	User can jump from one task to another and same time various application can run simultaneously.
Resizable widgets	Widgets are resizable, so users can expand them to show more content or shrink them to save space
Multi-Language	Supports single direction and bi-directional text.
GCM	Google Cloud Messaging (GCM) is a service that lets developers send short message data to their users on Android devices, without needing a proprietary sync solution.
Wi-Fi Direct	A technology that lets apps discover and pair directly, over a high-bandwidth peer-to-peer connection.
Android Beam	A popular NFC-based technology that lets users instantly share, just by touching two NFC-enabled phones together.

Android Applications

Android applications are usually developed in the Java language using the Android Software Development Kit. Once developed, Android applications can be packaged easily and sold out either through a store such as **Google Play**, **SlideME**, **Opera Mobile Store**, **Mobango**, **F-droid** and the **Amazon App store**.

Code name	Version number	Initial release date	API level
	1.0	September 23, 2008	1
	1.1	February 9, 2009	2

Cupcake	1.5	April 27, 2009	3
Donut	1.6	September 15, 2009	4
Eclair	2.0–2.1	October 26, 2009	5–7
Froyo	2.2–2.2.3	May 20, 2010	8
Gingerbread	2.3–2.3.7	December 6, 2010	9–10
Honeycomb^[a]	3.0–3.2.6	February 22, 2011	11–13
Ice Cream Sandwich	4.0–4.0.4	October 18, 2011	14–15
Jelly Bean	4.1–4.3.1	July 9, 2012	16–18
KitKat	4.4–4.4.4, 4.4W–4.4W.2	October 31, 2013	19–20
Lollipop	5.0–5.1.1	November 12, 2014	21–22
Marshmallow	6.0–6.0.1	October 5, 2015	23

[N](#) *Developer Preview 4*

API Levels

In computer programming, an **application programming interface (API)** is a set of routine definitions, protocols, and tools for building software and applications. An API may be for a web-based system, operating system, or database system, and it provides facilities to develop applications for that system using a given programming language. an API can facilitate integration of new features into existing applications.

In most procedural languages, an API specifies a set of [functions or routines](#) that accomplishes a specific task, or are allowed to interact with a specific software component. the math API on [Unix](#) systems is a specification on how to use the mathematical functions included in the math library. Among these functions there is a function named `sqrt()`, that can be used to compute the square root of a given number.

The Unix command `man 3 sqrt` presents the [signature](#) of the function `sqrt` in the form:

SYNOPSIS

```
#include <math.h>
double sqrt(double X);
float sqrtf(float X);
```

DESCRIPTION

`sqrt` computes the positive square root of the argument. ...

RETURNS

On success, the square root is returned. If `X` is real and positive.

Hence the API in this case can be interpreted as the collection of the [include files](#) used by a program, written in the C language, to reference that library function, and its human readable description provided by the [man pages](#).

Each Android device supports exactly *one* API level. Android's system of API levels helps Android determine whether an application is compatible with an Android system image prior to installing the application on a device. If the API level of an Android device is lower than the minimum API level that you specify for your app, the Android device will prevent the user from installing your app.

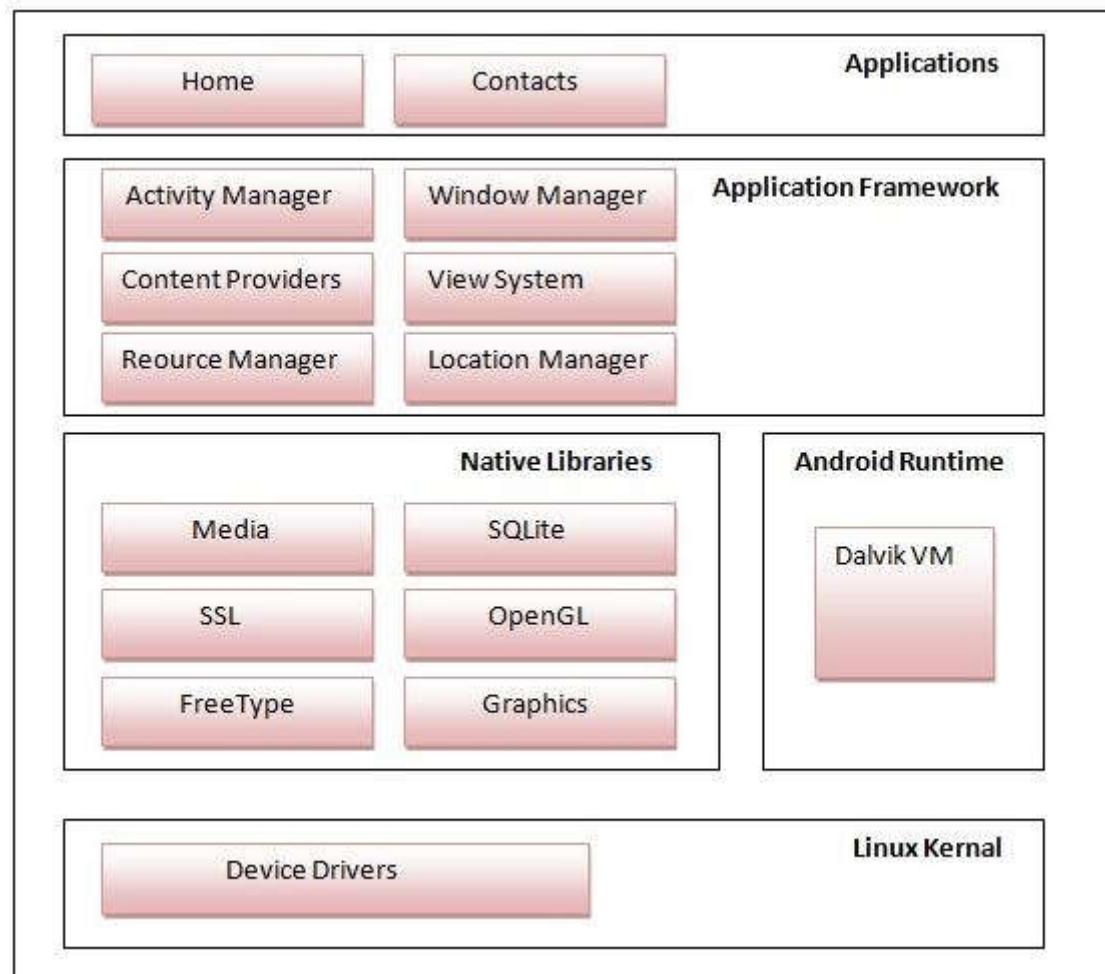
- The *target* API level of Android that the app is built to run on.
- The *minimum* API level of Android that is required to run the app.

You can set target APIs as well as Minimum APIs for application, in application settings

Popular API Examples

Twitter, Google map APIs,

Android Architecture



Linux Kernel

It is the heart of android architecture that exists at the root of android architecture. **Linux kernel** is responsible for device drivers, power management, memory management, device management and resource access, network management (IPC).

Libraries

It includes set of libraries including open-source Web browser engine WebKit, well known library libc, OpenGL (Open Graphics Library) for GPU, SQLite database which is a useful repository for storage and sharing of application data, libraries to play and record audio and video, SSL libraries responsible for Internet security etc.

Android Runtime

In android runtime, there are core libraries and DVM (Dalvik Virtual Machine) which is responsible to run android application. DVM is like JVM but it is optimized for mobile devices. It consumes less memory and provides fast performance. The Dalvik VM enables every Android application to run in its own process, with its own instance of the Dalvik virtual machine (.DEX file). Android programs are compiled into .dex (Dalvik Executable) files, which are in turn zipped into a single .apk file on the device. .dex files can be created by automatically translating compiled applications written in the Java programming language. The Android runtime

also provides a set of core libraries which enable Android application developers to write Android applications using standard Java programming language.

Android Framework

Android framework includes **Android API's** such as UI (User Interface), telephony, resources, locations, Content Providers (data) and package managers. It provides a lot of classes and interfaces for android application development.

- **Activity Manager** – Controls all aspects of the application lifecycle and activity stack. (All Activity)
- **Content Providers** – Allows applications to publish and share data with other applications. (Permission)
- **Resource Manager** – Provides access to non-code embedded resources such as strings, color settings and user interface layouts.
- **Notifications Manager** – Allows applications to display alerts and notifications to the user.
- **View System** – An extensible set of views used to create application user interfaces.

Applications

- On the top of android framework, there are applications. All applications such as home, contact, settings, games, browsers are using android framework that uses android runtime and libraries, Android runtime and native libraries are using Linux kernel.

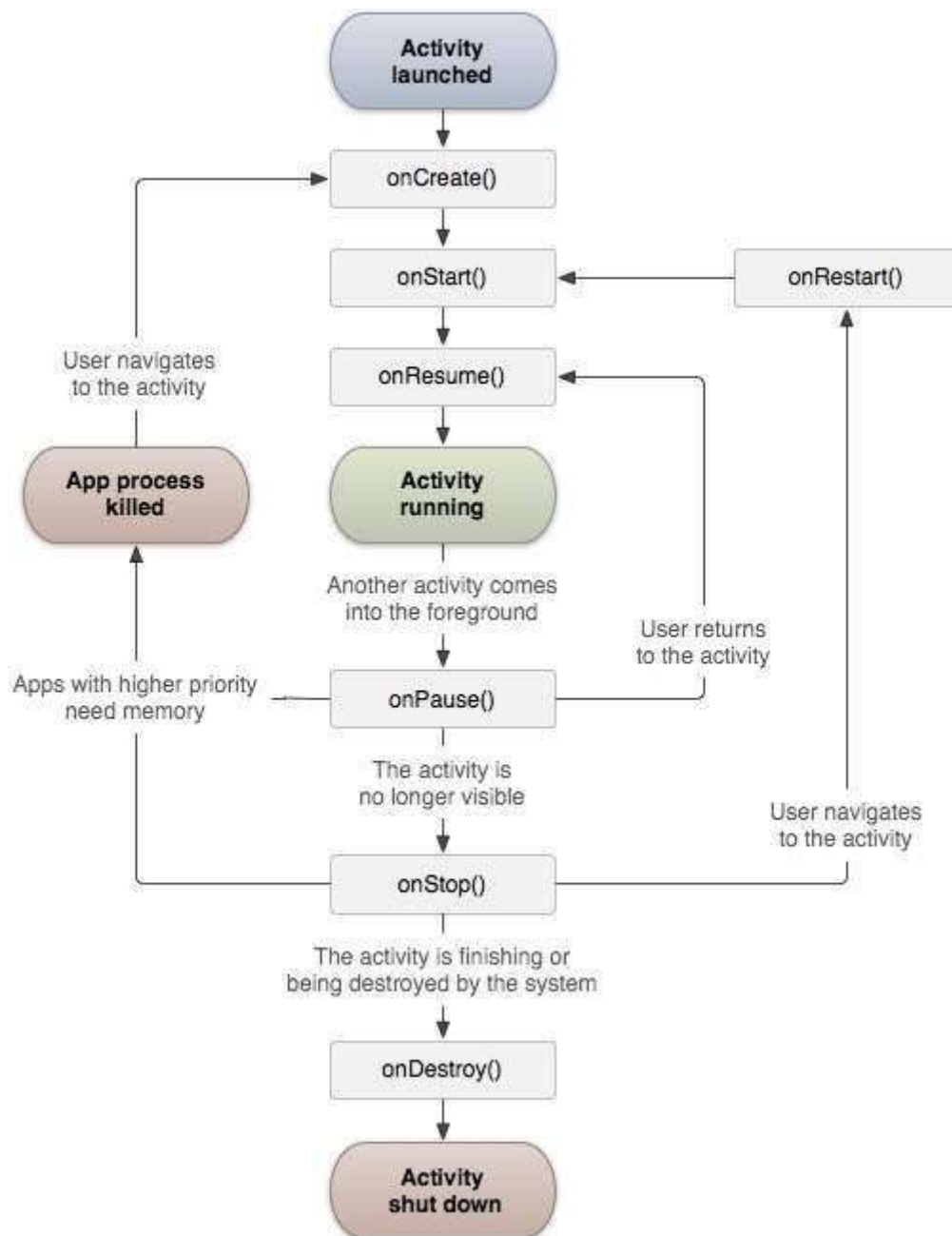
Android - Application Components

There are following four main components that can be used within an Android application:

Components	Description
Activities	They dictate the UI and handle the user interaction to the smart phone screen
Services	They handle background processing associated with an application.
Broadcast Receivers	They handle communication between Android OS and applications.
Content Providers	They handle data and database management issues.

Activities

An activity represents a single screen with a user interface,in-short Activity performs actions on the screen. For example, an email application might have one activity that shows a list of new emails (inbox), another activity to compose an email, and another activity for reading emails. If an application has more than one activity, then one of them should be marked as the activity that is presented when the application is launched.



The Activity class defines the following call backs i.e. events. You don't need to implement all the callbacks methods.

Callback	Description
onCreate()	This is the first callback and called when the activity is first created.
onStart()	This callback is called when the activity becomes visible to the user.
onResume()	This is called when the user starts interacting with the application.
onPause()	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.
onStop()	This callback is called when the activity is no longer visible.
onDestroy()	This callback is called before the activity is destroyed by the system.
onRestart()	This callback is called when the activity restarts after stopping it.

```

public class MainActivity extends Activity {

    @Override

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main); //Load UI as in activity_main.xml
    }
}

```

An application can have one or more activities without any restrictions. Every activity you define for your application must be declared in your *AndroidManifest.xml* file and the main activity for your app must be declared in the manifest with an `<intent-filter>`.

AndroidManifest.xml

```

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.helloworld"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="22" />

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

        <activity
            android:name=".MainActivity"
            android:label="@string/title_activity_main" >

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

        </activity>

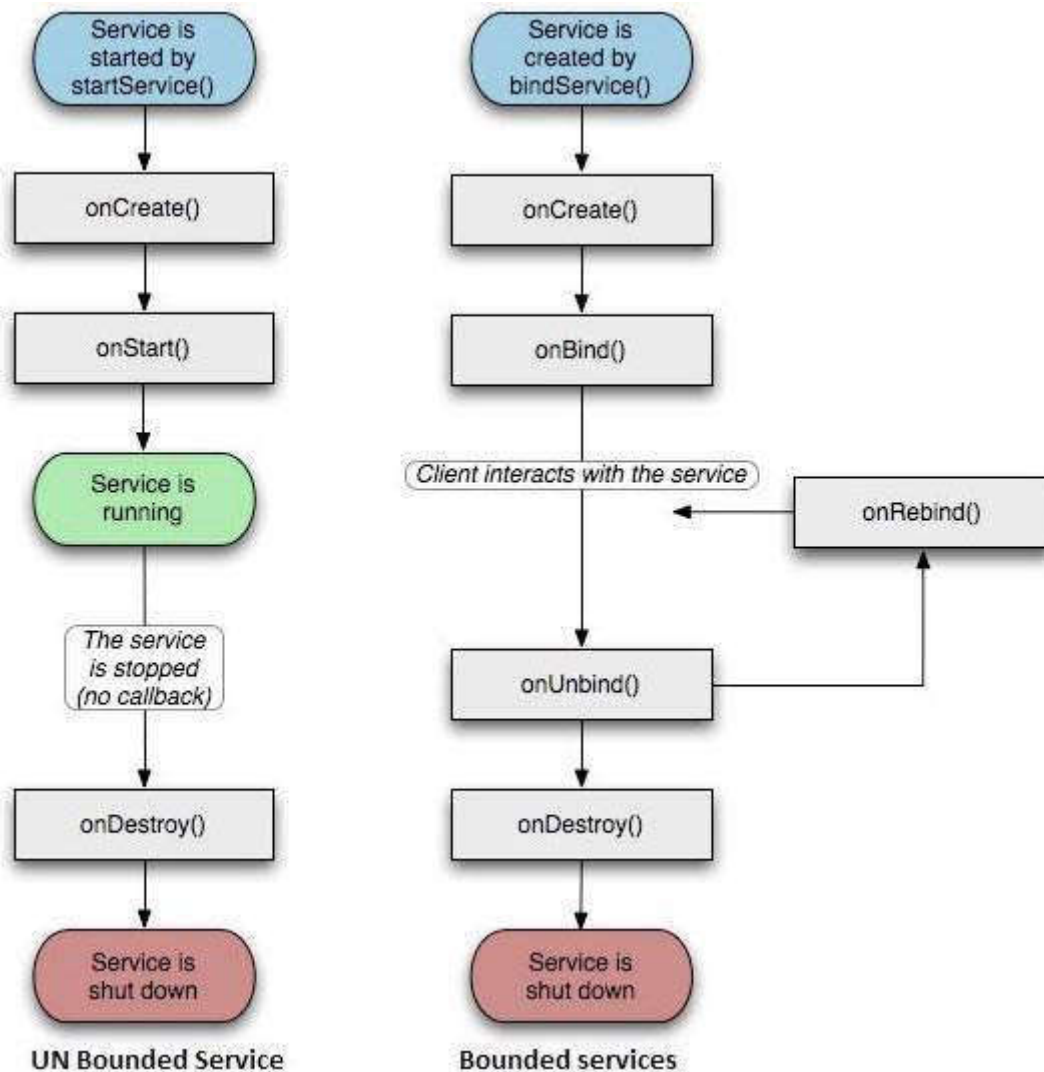
    </application>
</manifest>

```

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.

State	Description
Started	A service is started when an application component, such as an activity, starts it by calling <i>startService()</i> . Once started, a service can run in the background indefinitely, even if the component that started it is destroyed.
Bound	A service is bound when an application component binds to it by calling <i>bindService()</i> . 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).



To create a 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

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. It is a **passive data structure holding an abstract description of an operation to be performed**.

For example, 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 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.

Components:

1) Action

This is mandatory part of the Intent object and is a string naming the action to be performed. 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. The action in an Intent object can be set by the `setAction()` method and read by `getAction()`.

ACTION_ALL_APPS

List all the applications available on the device.

ACTION_ANSWER

Handle an incoming phone call.

ACTION_MAIN

Start as a main entry point, does not expect to receive data.

Data

Adds a data specification to an intent filter. The specification can be just a data type (the `mimeType` 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()`.

ACTION_VIEW `content://contacts/people/1`

Display information about the person whose identifier is "1".

ACTION_SYNC

It going to be synchronous the data, Constant Value is **`android.intent.action.SYNC`**

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.

CATEGORY_LAUNCHER

Should be displayed in the top-level launcher.

CATEGORY_APP_BROWSER

Used with **ACTION_MAIN** to launch the browser application.

CATEGORY_APP_CALCULATOR

Used with **ACTION_MAIN** to launch the calculator application.

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.

EXTRA_SUBJECT

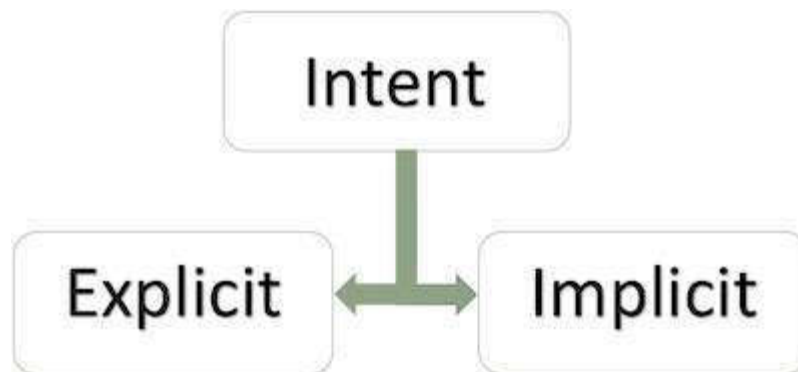
A constant string holding the desired subject line of a message.

EXTRA_EMAIL

A String[] holding e-mail addresses that should be delivered to.

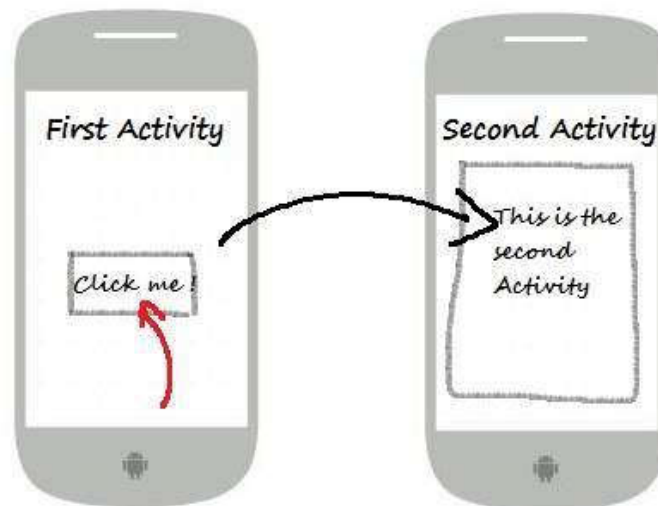
Types of Intents

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);
```

Explicit intents specify the component to start by name (the fully-qualified class name). You'll typically use an explicit intent to start a component in your own app, because you know the class name of the activity or service you want to start. For example, start a new activity in response to a user action or start a service to download a file in the background.

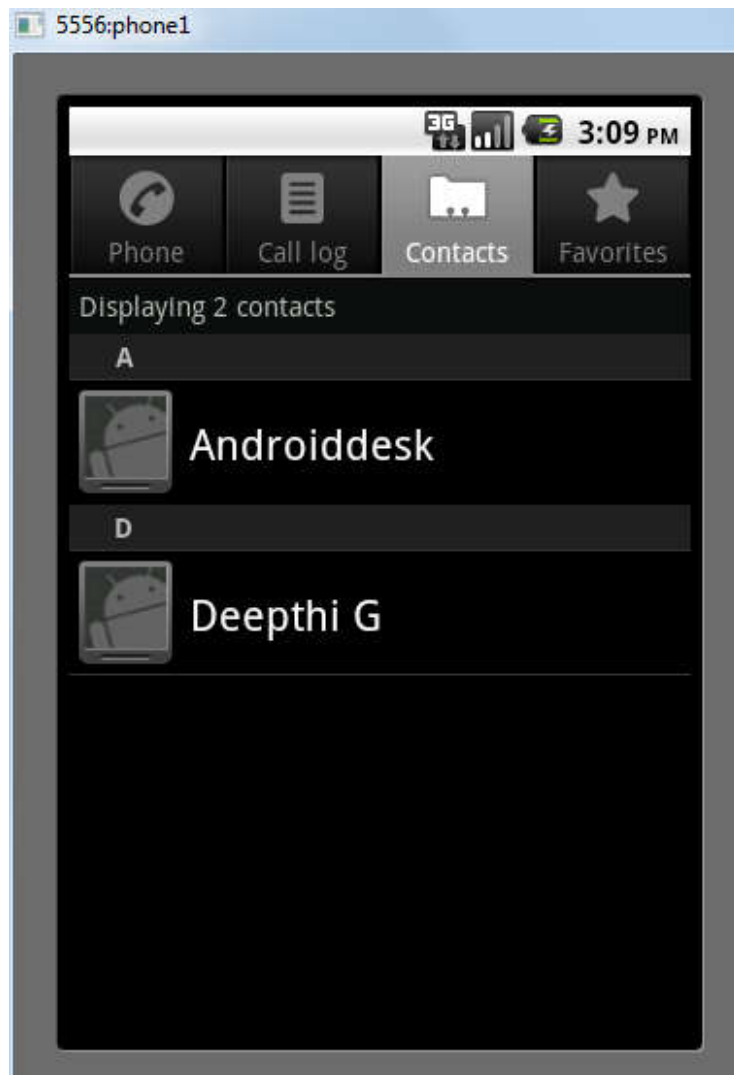
Implicit Intents

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.

Implicit intents do not name a specific component, but instead declare a general action to perform, which allows a component from another app to handle it. For example, if you want to show the user a location on a map, you can use an implicit intent to request that another capable app show a specified location on a map.

For example –

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



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");
```

ACTIVITY Vs Fragments

An [Activity](#) is an application component that provides a screen with which users can interact in order to do something, such as dial the phone, take a photo, send an email, or view a map. Each activity is given a window in which to draw its user interface. The window typically fills the screen, but may be smaller than the screen and float on top of other windows.

A [Fragment](#) represents a behavior or a portion of user interface in an [Activity](#). You can combine multiple fragments in a single activity to build a multi-pane UI and reuse a fragment in multiple activities. You can think of a fragment as a modular section of an activity, which has its own lifecycle, receives its own input events, and which you can add or remove while the activity is running (sort of like a "sub activity" that you can reuse in different activities).

Android - Broadcast Receivers

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 step in case you are going to implement your custom intents then you will have to create and broadcast those intents.

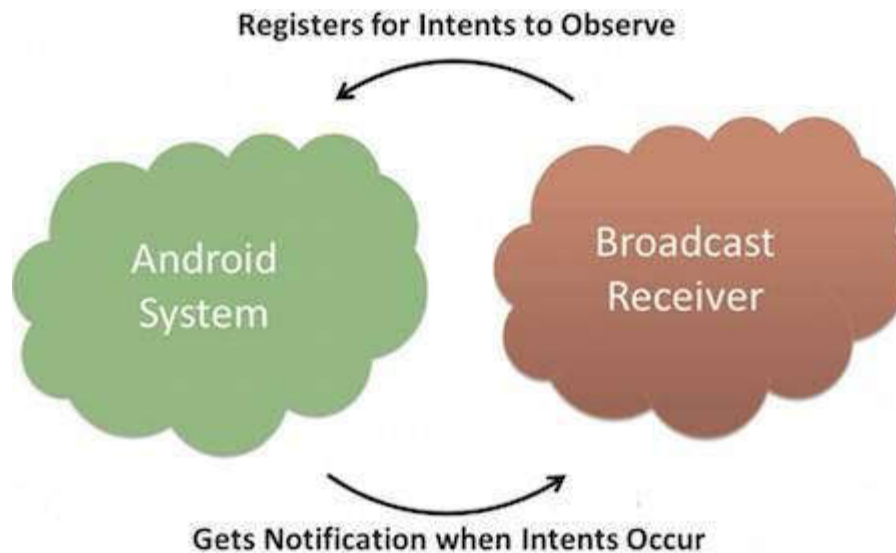
Creating the Broadcast Receiver

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.



Broadcast-Receiver

```
<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.

Event Constant	Description
android.intent.action.BATTERY_CHANGED	Sticky broadcast containing the charging state, level, and other information about the battery.
android.intent.action.BATTERY_LOW	Indicates low battery condition on the device.
android.intent.action.BATTERY_OKAY	Indicates the battery is now okay after being low.
android.intent.action.BOOT_COMPLETED	This is broadcast once, after the system has finished booting.
android.intent.action.BUG_REPORT	Show activity for reporting a bug.
android.intent.action.CALL	Perform a call to someone specified by the data.
android.intent.action.CALL_BUTTON	The user pressed the "call" button to go to the dialer or other appropriate UI for placing a call.
android.intent.action.DATE_CHANGED	The date has changed.
android.intent.action.REBOOT	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.tutorialspoint.CUSTOM_INTENT");
    sendBroadcast(intent);
}
```

This intent `com.tutorialspoint.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.tutorialspoint.CUSTOM_INTENT">
            </action>
        </intent-filter>
    </receiver>
</application>
```

Example

This example will explain you how to create *BroadcastReceiver* to intercept custom intent. Once you are familiar with custom intent, then you can program your application to intercept system generated intents. So let's follow the following steps to modify the Android application we created in *Hello World Example*–

Step	Description
1	You will use Android studio to create an Android application and name it as <i>My Application</i> under a package <i>com.example.My Application</i> .
2	Modify main activity file <i>MainActivity.java</i> to add <i>broadcastIntent()</i> method.
3	Create a new java file called <i>MyReceiver.java</i> under the package <i>com.example.My Application</i> to define a <i>BroadcastReceiver</i> .
4	An application can handle one or more custom and system intents without any restrictions. Every intent you want to intercept must be registered in your <i>AndroidManifest.xml</i> file using <code><receiver.../></code> tag
5	Modify the default content of <i>res/layout/activity_main.xml</i> file to include a button to broadcast intent.
6	No need to modify the string file, Android studio take care of string.xml file.
7	Run the application to launch Android emulator and verify the result of the changes done in the application.

Following is the content of the modified main activity file **src/com.example.My Application/MainActivity.java**. This file can include each of the fundamental life cycle methods. We have added *broadcastIntent()* method to broadcast a custom intent.

```
package com.example.My Application;
import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.content.Intent;
import android.view.View;

public class MainActivity extends Activity {

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

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.activity_main, menu);
        return true;
    }

    // broadcast a custom intent.
    public void broadcastIntent(View view){
        Intent intent = new Intent();
        intent.setAction("com.tutorialspoint.CUSTOM_INTENT");
        sendBroadcast(intent);
    }
}
```

Following is the content of **src/com.example.My Application/MyReceiver.java**:

```
package com.example.My Application;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.widget.Toast;

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

Following will the modified content of *AndroidManifest.xml* file. Here we have added `<service.../>` tag to include our service:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.My Application"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="22" />
```

```

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

    <activity
        android:name=".MainActivity"
        android:label="@string/title_activity_main" >
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER"/>
        </intent-filter>
    </activity>

    <receiver android:name="MyReceiver">
        <intent-filter>
            <action android:name="com.tutorialspoint.CUSTOM_INTENT">
                </action>
            </intent-filter>
        </receiver>
    </application>
</manifest>

```

Following will be the content of **res/layout/activity_main.xml** file to include a button to broadcast our custom intent –

```

<RelativeLayout 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" android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin" tools:context=".MainActivity">

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Example of Broadcast"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:textSize="30dp" />

    <TextView
        android:id="@+id/textView2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Tutorials point "
        android:textColor="#ff87ff09"
        android:textSize="30dp"
        android:layout_above="@+id/imageButton"
        android:layout_centerHorizontal="true"
        android:layout_marginBottom="40dp" />

    <ImageButton
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/imageButton"
        android:src="@drawable/abc"
        android:layout_centerVertical="true"

```

```

        android:layout_centerHorizontal="true" />

<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/button2"
    android:text="Broadcast Intent"
    android:onClick="broadcastIntent"
    android:layout_below="@+id/imageButton"
    android:layout_centerHorizontal="true" />

</RelativeLayout>

```

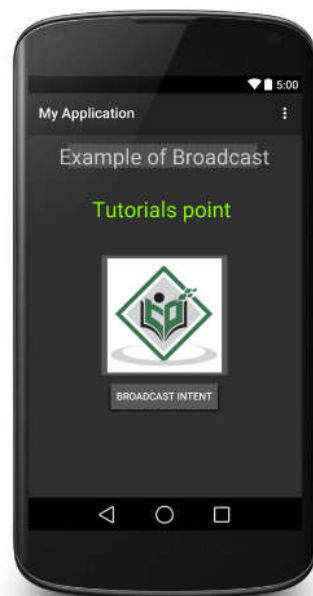
Following will be the content of **res/values/strings.xml** to define two new constants –

```

<resources>
    <string name="menu_settings">Settings</string>
    <string name="title_activity_main">My Application</string>
</resources>

```

When you run the application, it will display following Emulator window



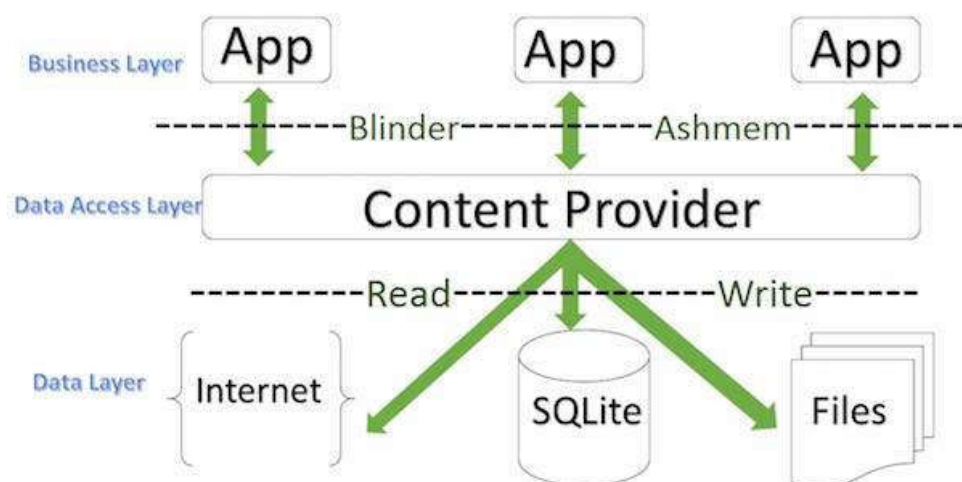
Now to broadcast our custom intent, let's click on **Broadcast Intent** button, this will broadcast our custom intent *"com.tutorialspoint.CUSTOM_INTENT"* which will be intercepted by our registered BroadcastReceiver i.e. MyReceiver and as per our implemented logic a toast will appear on the bottom of the the simulator as follows –



You can try implementing other BroadcastReceiver to intercept system generated intents like system boot up, date changed, low battery etc.

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.



Sometimes it is required to share data across applications. This is where content providers become very useful.

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.

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>

Part	Description
Prefix	This is always set to content://
authority	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>
data_type	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>
Id	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> .

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 (code file) or you can declare your layout using simple XML file **main_layout.xml** which is located in the **res/layout** folder of your project.

The advantage to declaring your UI in XML is that it enables you to better separate the presentation of your application from the code that controls its behaviour. Your UI descriptions are external to your application code, which means that you can modify or adapt it without having to modify your source code and recompile. For example, you can create XML layouts for different screen orientations, different device screen sizes, and different languages. Additionally, declaring the layout in XML makes it easier to visualize the structure of your UI, so it's easier to debug problems.

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.

Layout	Description
Linear Layout	LinearLayout is a view group that aligns all children in a single direction, vertically or horizontally.
Relative Layout	RelativeLayout is a view group that displays child views in relative positions.
Table Layout	TableLayout is a view that groups views into rows and columns.
Absolute Layout	AbsoluteLayout enables you to specify the exact location of its children.
Frame Layout	The FrameLayout is a placeholder on screen that you can use to display a single view.
List View	ListView is a view group that displays a list of scrollable items.
Grid View	GridView is a ViewGroup that displays items in a two-dimensional, scrollable grid.

Layout Attributes

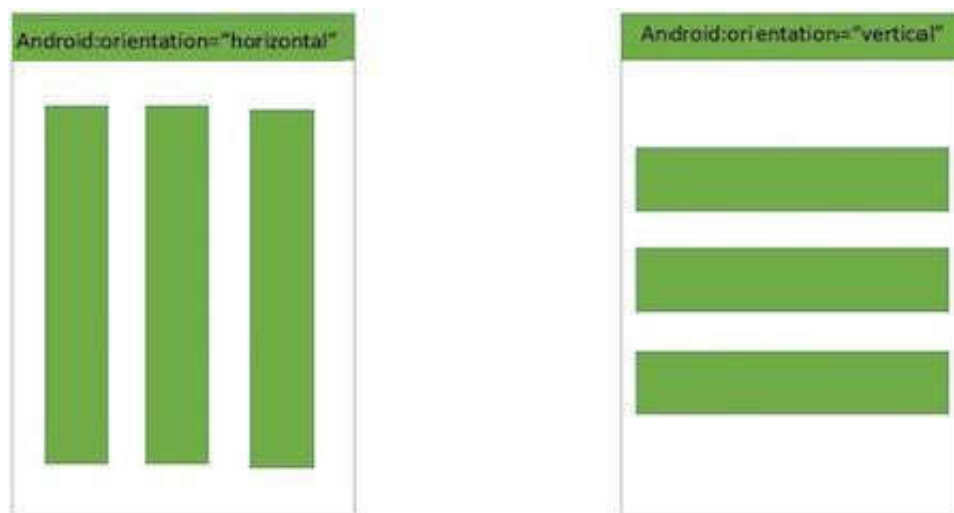
Each layout has a set of attributes which define the visual properties of that layout. There are few common attributes among all the layouts and there are other attributes which are specific to that layout. Following are common attributes and will be applied to all the layouts:

Attribute	Description
android:id	This is the ID which uniquely identifies the view.
android:layout_width	This is the width of the layout.
android:layout_height	This is the height of the layout
android:layout_marginTop	This is the extra space on the top side of the layout.
android:layout_marginBottom	This is the extra space on the bottom side of the layout.
android:layout_marginLeft	This is the extra space on the left side of the layout.
android:layout_marginRight	This is the extra space on the right side of the layout.
android:layout_gravity	This specifies how child Views are positioned.
android:layout_weight	This specifies how much of the extra space in the layout should be allocated to the View.
android:layout_x	This specifies the x-coordinate of the layout.
android:layout_y	This specifies the y-coordinate of the layout.
android:layout_width	This is the width of the layout.
android:paddingLeft	This is the left padding filled for the layout.
android:paddingRight	This is the right padding filled for the layout.
android:paddingTop	This is the top padding filled for the layout.
android:paddingBottom	This is the bottom padding filled for the layout.

To ensure that your layout is flexible and adapts to different screen sizes, you should use "wrap_content" and "match_parent" for the width and height of some view components. If you use "wrap_content", the width or height of the view is set to the minimum size necessary to fit the content within that view (Best possible size to visible), while "match_parent" makes the component expand to match the size of its parent view.

LinearLayout

Android LinearLayout is a view group that aligns all children in either *vertically* or *horizontally*. You can specify the layout direction with the [android:orientation](#) attribute.



Attribute	Description
android:id	This is the ID which uniquely identifies the layout.
android:baselineAligned	This must be a boolean value, either "true" or "false" and prevents the layout from aligning its children's baselines.
android:baselineAlignedChildIndex	When a linear layout is part of another layout that is baseline aligned, it can specify which of its children to baseline align
android:divider	This is drawable to use as a vertical divider between buttons. You use a color value, in the form of "#rgb", "#argb", "#rrggbb", or "#aarrggbb".
android:gravity	This specifies how an object should position its content, on both the X and Y axes. Possible values are top, bottom, left, right, center, center_vertical, center_horizontal etc.
android:orientation	This specifies the direction of arrangement and you will use "horizontal" for a row, "vertical" for a column. The default is horizontal.
android:weightSum	Sum up of child weight

Android Relative Layout

Android RelativeLayout enables you to specify how child views are positioned relative to each other. The position of each view can be specified as relative to sibling elements or relative to the parent.

Attribute	Description
android:id	This is the ID which uniquely identifies the layout.
android:gravity	This specifies how an object should position its content, on both the X and Y axes. Possible values are top, bottom, left, right, center, center_vertical, center_horizontal etc.
android:ignoreGravity	This indicates what view should not be affected by gravity.



Using `RelativeLayout`, you can align two elements by right border, or make one below another, centered in the screen, centered left, and so on.

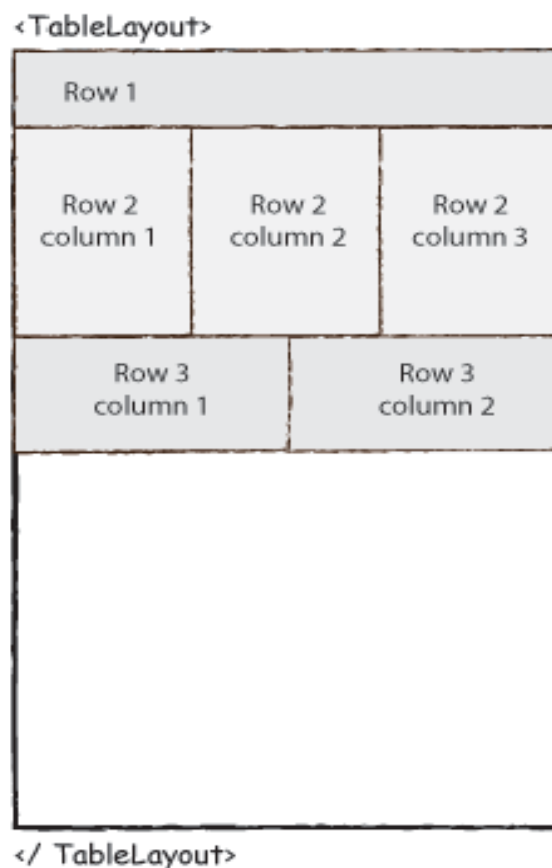
By default, all child views are drawn at the top-left of the layout, so you must define the position of each view using the various layout properties available from `RelativeLayout.LayoutParams`.

The ScrollView Layout

- Most Android application will likely to have the contents that's doesn't fit the screen.
- For Example, displaying the news details, the contents are dynamic and can grow beyond your screen size. If we design our screen layout using standard layout managers like [LinearLayout](#), [RelativeLayout](#), [FrameLayout](#), [TableLayout](#); when the content grows, and data goes beyond screen size, and user won't be able scroll and view the content.
- **ScrollView** is a special kind of layout, designed to hold view larger than its actual size. When the Views size goes beyond the ScrollView size, it automatically adds scroll bars and can be scrolled vertically.
- You can specify `layout_height` and `layout_width` to adjust height and width of screen.
- `android:fillViewport`: "Defines whether the scrollview should stretch its content to fill the viewport." Either "true" or "false".
- In certain situations, you want to position content beneath the end of the scrollable content area. For example for a "terms of service" where you can only accept once you've scrolled through all the content. In this case, you might need to apply the [android:fillViewport](#)

Android Table Layout

- Android `TableLayout` going to be arranged groups of views into rows and columns. You will use the `<TableRow>` element to build a row in the table.
- Each row have more than one cells;
- each cell can hold one View object.



TableLayout Attributes

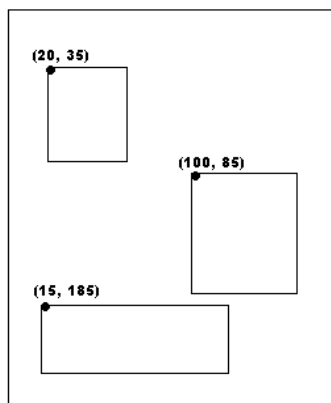
Following are the important attributes specific to TableLayout –

Attribute	Description
android:id	This is the ID which uniquely identifies the layout.
android:collapseColumns	This specifies the zero-based index of the columns to collapse. The column indices must be separated by a comma: 1, 2, 5.
android:shrinkColumns	<p>The zero-based index of the columns to shrink. The column indices must be separated by a comma: 1, 2, 5. If the value is 1 then the second column is stretched to take up any available space in the row, because of the column numbers are started from 0. If the value is 0,1 then both the first and second columns of table are stretched to take up the available space in the row.</p> <p>If the value is '*' then all the columns are stretched to take up the available space.</p>
android:stretchColumns	The zero-based index of the columns to stretch. The column indices must be separated by a comma: 1, 2, 5.

Android Absolute Layout

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

Absolute Layout



Android TextView Control

- A TextView displays text to the user and optionally allows them to edit it.

TextView Attributes

Attribute	Description
android:id	This is the ID which uniquely identifies the control.
android:capitalize	<ul style="list-style-type: none">• If set, specifies that this TextView has a textual input method and should automatically capitalize what the user types. Don't automatically capitalize anything - 0• Capitalize the first word of each sentence - 1• Capitalize the first letter of every word - 2• Capitalize every character – 3
android:cursorVisible	Makes the cursor visible (the default) or invisible. Default is false.
android:editable	If set to true, specifies that this TextView has an input method.
android:fontFamily	Font family (named by string) for the text.
android:gravity	Specifies how to align the text by the view's x- and/or y-axis when the text is smaller than the view.
android:hint	Hint text to display when the text is empty.
android:inputType	The type of data being placed in a text field. Phone, Date, Time, Number, Password etc.
android:maxHeight	Makes the TextView be at most this many pixels tall.
android:maxLength	Makes the TextView be at most this many pixels wide.
android:minHeight	Makes the TextView be at least this many pixels tall.
android:minWidth	Makes the TextView be at least this many pixels wide.
android:password	Whether the characters of the field are displayed as password dots instead of themselves. Possible value either "true" or "false".
android:phoneNumber	If set, specifies that this TextView has a phone number input method. Possible value either "true" or "false".
android:text	Text to display.
android:textAllCaps	Present the text in ALL CAPS. Possible value either "true" or "false".
android:textColor	Text color. May be a color value, in the form of "#rgb", "#argb", "#rrggbb", or "#aarrggbb".
android:textColorHighlight	Color of the text selection highlight.
android:textColorHint	Color of the hint text. May be a color value, in the form of "#rgb", "#argb", "#rrggbb", or "#aarrggbb".
android:textIsSelectable	Indicates that the content of a non-editable text can be selected. Possible value either "true" or "false".
android:textSize	Size of the text. Recommended dimension type for text is "sp" for scaled-pixels (example: 15sp).
android:textStyle	<ul style="list-style-type: none">• Style (bold, italic, bolditalic) for the text. You can use or more of the following values separated by ' '.normal - 0• bold - 1• italic – 2
android:typeface	<ul style="list-style-type: none">• Typeface (normal, sans, serif, monospace) for the text. You can use or more of the following values separated by ' '.normal - 0• sans - 1• serif - 2• monospace – 3

Android EditText Control

A EditText is an overlay over TextView that configures itself to be editable. It is the predefined subclass of TextView that includes rich editing capabilities.

EditText Attribute

Attribute	Description
android:autoText	If set, specifies that this TextView has a textual input method and automatically corrects some common spelling errors.
android:drawableBottom	This is the drawable to be drawn below the text. It can be image, color.
android:drawableRight	This is the drawable to be drawn to the right of the text.
android:editable	If set, specifies that this TextView has an input method.
android:text	This is the Text to display.

Button Widget

- A Button is a Push-button which can be pressed, or clicked, by the user to perform an action.



Handling Clicking Events

1) By using Listener for every Button

- You create the button
- You create an instance of OnClickListener and override the onClick-method.
- You assign that OnClickListener to that button using `btn.setOnClickListener(myOnClickListener);` in your fragments/activities onCreate-method.
- When the user clicks the button, the onClick function of the assigned OnClickListener is called.

Code:

```

public class MainActivity extends ActionBarActivity
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main); // Capture our button from layout
        Button button = (Button)findViewById(R.id.corky);
        button.setOnClickListener(mCorkyListener);
    }
    private View.OnClickListener mCorkyListener = new View.OnClickListener()
    {
        public void onClick(View v)
        {
            // do something when the button is clicked //
        }
    };
}

```

You can create Listener for more than one Button in single Listener as follow

```

Button button = (Button)findViewById(R.id.corky);
Button button2 = (Button)findViewById(R.id.corky2);
Button button3 = (Button)findViewById(R.id.corky3);
button.setOnClickListener(mCorkyListener); button2.setOnClickListener(mCorkyListener);
button3.setOnClickListener(mCorkyListener);
private View.OnClickListener mCorkyListener = new View.OnClickListener(){
    public void onClick(View v) {
        switch (v.getId() /*to get clicked view id*/)
        {
            case R.id.corky: // do something ( for Button with id corky)
                break;
            case R.id.corky2: // do something when the corky2 is clicked (Button with id corky2)
                break;
            case R.id.corky3: // do something when the corky3 is clicked (Button with id
corky3)      break;
            default:
                break;
        }
    }
};

```

You can also do same thing by using **Implements keyword** at the starting

Code:-

```

public class MainActivity extends ActionBarActivity implements View.OnClickListener
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    { super.onCreate(savedInstanceState); setContentView(R.layout.activity_main);
    Button button = (Button)findViewById(R.id.corky);
    Button button2 = (Button)findViewById(R.id.corky2);
    // Register the onClick listener with the implementation above button.setOnClickListener(this);
    button2.setOnClickListener(this);
    }
    @Override
    public void onClick(View v)
    {
        switch (v.getId() /*to get clicked view id*/)

```

```

        {
            case R.id.corky: // do something when the corky is clicked
                break;
            case R.id.corky2: // do something when the corky2 is clicked
                break;
            Default:
                break;
        }
    }
}

```

2) By using android:onClick attribute

In Layout file (e.g. android_main.xml) add **android:onClick** attribute in Button Tag

```

<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Register"
    android:id="@+id/Register_button"
    android:onClick="Register"/>

```

In java file (e.g. MainActivity.java) add following code

```

protected void Register(View v)
{
    // do implements your logic here
}

```

RadioButton

- A RadioButton has two states: either checked or unchecked.
- This allows the user to select one option from a set.
- If we check one radio button that belongs to a radio group, it automatically uncheck any previously checked radio button within the same group.



Important Attribute

- Android:checked="T/F" {To Set RadioButton option true (Selected) or False (unselected)}
- Android:onClick="Method Name" {For Implementing option checking Logic }
- getCheckedRadioButtonId() method

Following Example shows use of RadioButton OnClick method Implementation using Listener:

- **rb1**=(RadioButton)findViewById(R.id.*radioButton*);
- **rb2**=(RadioButton)findViewById(R.id.*radioButton2*);

rb1.setOnClickListener(**new** View.OnClickListener()

```

{
    @Override
    public void onClick(View v)

```

```

{

    Toast.makeText(Radio_Button.this,rb1.getText().toString(),Toast.LENGTH_LONG).show();

    //This will display pop up message for
    });

    rb2.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            Toast.makeText(Radio_Button.this,rb2.getText().toString(),Toast.LENGTH_LONG).show();
        }
    });
}

```

ImageButton

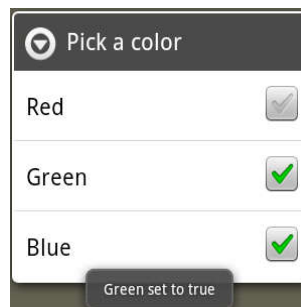
- Displays a button with an image (instead of text) that can be pressed or clicked by the user.
- By default, an ImageButton looks like a regular [Button](#), with the standard button background that changes color during different button states.
- android:src – This Attribute sets a drawable as the content of this ImageView.

For e.g. android:src="@drawable/abc.png"

- android:onClick This is the name of the method in this View's context to invoke when the view is **clicked**.

CheckBox

- A CheckBox is an on/off switch that can be toggled by the user.
- You should use check-boxes when presenting users with a group of selectable options that are not mutually exclusive. For example person's hobbies.



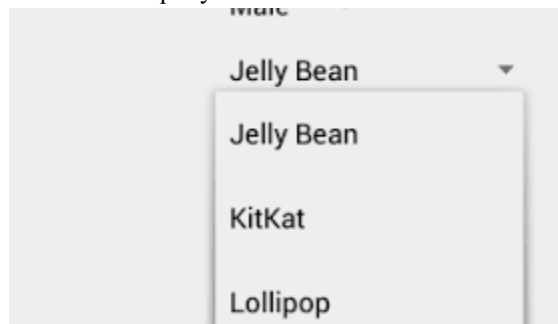
Attributes

- Android:onClick- This is the name of the method in this View's context to invoke when the view is clicked.
- isChecked()- To check whether checkbox is checked or not. (True/False)
- setChecked()- To set checkbox checked (True) or unchecked (false)

Spinner

- Spinners provide a quick way to select one value from a set.
- Touching the spinner displays a dropdown menu with all other available values, from which the user can select a new one.

- The choices you provide for the spinner can come from any source, but must be provided through an [SpinnerAdapter](#), such as an [ArrayAdapter](#) if the choices are available in an array or a [CursorAdapter](#) if the choices are available from a database query.

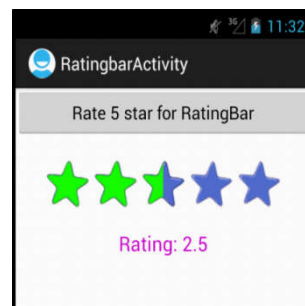


Steps to Load Item in Spinner

```
String[] country={"India","U.S.A.","U.K.","Paris","France","Ukrain"};
ArrayAdapter<String> country_name=new
ArrayAdapter<String>(this,android.R.layout.select_dialog_item,country);
Spinner spinner = (Spinner) findViewById(R.id.spinner);
adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
spinner.setAdapter(country_name);
```

RatingBar

- Android RatingBar** can be used to get the rating from the user.
- The Rating returns a floating-point number.
- It may be 2.0, 3.5, 4.0,4.5,5.0 etc.
- The **getRating()** method of android RatingBar class returns the **rating number** given by User.
- For example String rating=String.valueOf(ratingbar1.getRating());



Menu in android

Menus are a common user interface component in many types of applications. To provide a familiar and consistent user experience, you should use the [Menu](#) APIs to present user actions and other options in your activities.

Types of Menu

- Options menu and app bar**

The [options menu](#) is the primary collection of menu items for an activity. It's where you should place actions that have a global impact on the app, such as "Search," "Compose email," and "Settings."

- Context menu and contextual action mode**

A context menu is a [floating menu](#) that appears when the user performs a long-click on an element. It provides actions that affect the selected content or context frame. The [contextual action mode](#) displays action items that affect the selected content in a bar at the top of the screen and allows the user to select multiple items.

- **Popup menu**

A popup menu displays a list of items in a vertical list that's anchored to the view that invoked the menu. It's good for providing an overflow of actions that relate to specific content or to provide options for a second part of a command. Actions in a popup menu should **not** directly affect the corresponding content—that's what contextual actions are for. Rather, the popup menu is for extended actions that relate to regions of content in your activity.

Defining a Menu in XML

For all menu types, Android provides a standard XML format to define menu items. Instead of building a menu in your activity's code, you should define a menu and all its items in an XML menu resource. You can then inflate the menu resource (load it as a Menu object) in your activity.

Using a menu resource is a good practice for a few reasons:

- It's easier to visualize the menu structure in XML.
- It separates the content for the menu from your application's behavioural code.
- It allows you to create alternative menu configurations for different platform versions, screen sizes, and other configurations by leveraging the [app resources](#) framework.

To define the menu, create an XML file inside your project's `res/menu/` directory and build the menu with the following elements

<menu>

Defines a Menu, which is a container for menu items. A <menu> element must be the root node for the file and can hold one or more <item> and <group> elements.

<item>

Creates a MenuItem, which represents a single item in a menu. This element may contain a nested <menu> element in order to create a submenu.

<group>

An optional, invisible container for item elements. It allows you to categorize menu items so they share properties such as active state and visibility. For example, in editing option ->copy, paste, cut can be group together.

Exa_menu.xml file

```
• <menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/item1"
        android:title="@string/item1"
        android:icon="@drawable/group_item1_icon"
        android:showAsAction="ifRoom|withText"/>
    <group android:id="@+id/group">
        <item android:id="@+id/group_item1"
            android:onClick="onGroupItemClick"
            android:title="@string/group_item1"
            android:icon="@drawable/group_item1_icon" />
        <item android:id="@+id/group_item2"
            android:onClick="onGroupItemClick"
            android:title="@string/group_item2"
            android:icon="@drawable/group_item2_icon" />
    </group>
    <item android:id="@+id/submenu"
        android:title="@string/submenu_title"
        android:showAsAction="ifRoom|withText" >
```

```

        <menu>
            <item android:id="@+id/submenu_item1"
                android:title="@string/submenu_item1" />
        </menu>
    </item>
</menu>

```

- **android:id**

A resource ID that's unique to the item, which allows application to recognize the item when the user selects it.

- **android:icon**

A reference to a drawable to use as the item's icon.

- **android:title**

A reference to a string to use as the item's title.

- **android:onClick="Method name"**

The method to call when this menu item is clicked. The method must be declared in the activity as public and accept a [MenuItem](#) as its only parameter, which indicates the item clicked. This method takes precedence over the standard callback to [onOptionsItemSelected\(\)](#).

- **android:showAsAction**

Specifies when and how this item should appear as an action item in the app bar. The following table shows various possible values for this attributes

Value	Description
ifRoom	Only place this item in the app bar if there is room for it. If there is not room for all the items marked "ifRoom", the items with the lowest order In Category values are displayed as actions, and the remaining items are displayed in the overflow menu.
withText	Also include the title text (defined by android:title) with the action item. You can include this value along with one of the others as a flag set, by separating them with a pipe .
never	Never place this item in the app bar. Instead, list the item in the app bar's overflow menu.
always	Always place this item in the app bar. Avoid using this unless it's critical that the item always appear in the action bar. Setting multiple items to always appear as action items can result in them overlapping with other UI in the app bar.
collapseActionView	The action view associated with this action item (as declared by android:actionLayout or android:actionViewClass) is collapsible.

The following application code inflates the menu from the [onCreateOptionsMenu\(Menu\)](#) callback and also declares the on-click callback for two of the items:

```

public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater(); //used Intialize menu's xml file
    inflater.inflate(R.menu.example_menu, menu);
    return true;
}

public void onGroupItemClick(MenuItem item) {
    // One of the group items (using the onClick attribute) was clicked
    // The item parameter passed here indicates which item it is
    // All other menu item clicks are handled by onOptionsItemSelected()
}

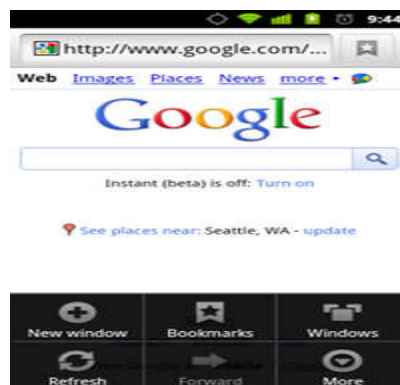
```

The options Menu

- The [options menu](#) is the primary collection of menu items for an activity.
- It's where you should place actions that have a global impact on the app, such as "Search," "Compose email," and "Settings."

Where the items in your options menu appear on the screen depends on the version for which you've developed your application:

- If you've developed your application for **Android 2.3.x (API level 10) or lower**, the contents of your options menu appear at the bottom of the screen when the user presses the *Menu* button, as shown in figure. When opened, the first visible portion is the icon menu, which holds up to six menu items. If your menu includes more than six items, Android places the sixth item and the rest into the overflow menu, which the user can open by selecting *more*.



- If you've developed your application for **Android 3.0 (API level 11) and higher**, items from the options menu are available in the app bar. By default, the system places all items in the action overflow, which the user can reveal with the action overflow icon on the right side of the app bar (or by pressing the device *Menu* button, if available). To enable quick access to important actions, you can promote a few items to appear in the app bar by adding `android:showAsAction="ifRoom"` to the corresponding `item` elements (see figure).



You can declare items for the options menu from your [Activity](#) subclass. To specify the options menu for an activity, override [onCreateOptionsMenu\(\)](#). In this method, you can inflate your menu resource ([defined in XML](#)) into the [Menu](#) provided in the callback.

Handling click events

When the user selects an item from the options menu (including action items in the app bar), the system calls your activity's [onOptionsItemSelected\(\)](#) method. This method passes the [MenuItem](#) selected. You can identify the item by calling [getItemId\(\)](#), which returns the unique ID for the menu item (defined by the `android:id` attribute in the menu resource or with an integer given to the [add\(\)](#) method). You can match this ID against known menu items to perform the appropriate action. For example:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.new_game:
            newGame();
            return true;
        case R.id.help:
            showHelp();
            return true;
        default:
            return super.onOptionsItemSelected(item);    }}
}
```

When you successfully handle a menu item, return `true`. If you don't handle the menu item, you should call the superclass implementation of [onOptionsItemSelected\(\)](#) (the default implementation returns false).

Changing menu items at runtime

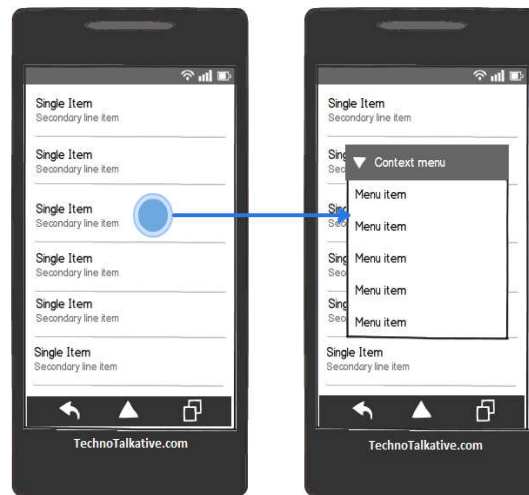
After the system calls [onCreateOptionsMenu\(\)](#), it retains an instance of the [Menu](#) you populate and will not call [onCreateOptionsMenu\(\)](#) again unless the menu is invalidated for some reason. However, you should use [onCreateOptionsMenu\(\)](#) only to create the initial menu state and not to make changes during the activity lifecycle. If you want to modify the options menu based on events that occur during the activity lifecycle, you can do so in the [onPrepareOptionsMenu\(\)](#) method. This method passes you the [Menu](#) object as it currently exists so you can modify it, such as add, remove, or disable items.

```
@Override
public boolean onPrepareOptionsMenu(Menu menu) {
    if(isStarted) {
        menu.removeItem(R.id.start);
        menu.add(R.id.stop);
    } else {
        menu.removeItem(R.id.stop);
        menu.add(R.id.start);
    }
    return true;
}
```

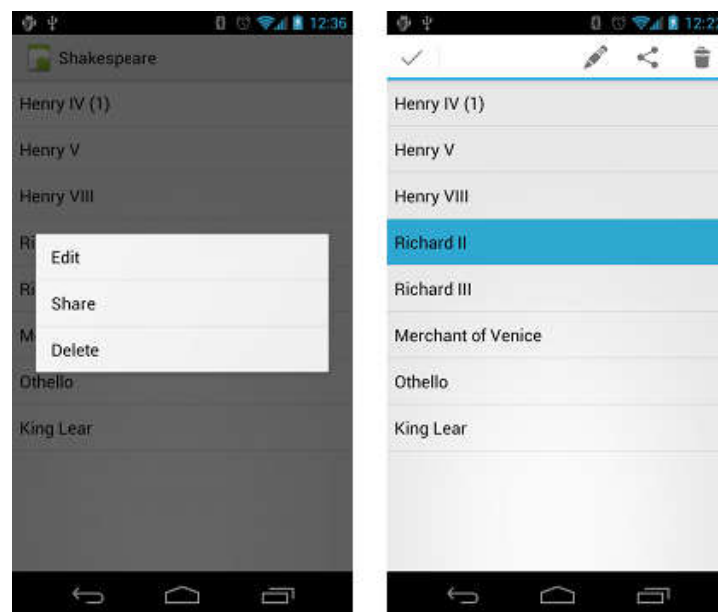
On Android 2.3.x and lower, the system calls [onPrepareOptionsMenu\(\)](#) each time the user opens the options menu (presses the *Menu* button). On Android 3.0 and higher, the options menu is considered to always be open when menu items are presented in the app bar. When an event occurs and you want to perform a menu update, you must call [invalidateOptionsMenu\(\)](#) to request that the system call [onPrepareOptionsMenu\(\)](#).

Creating Contextual Menus

A contextual menu offers actions that affect a specific item or context frame in the UI. You can provide a context menu for any view, but they are most often used for items in a [ListView](#), [GridView](#), or other view collections in which the user can perform direct actions on each item.



- In a [floating context menu](#). A menu appears as a floating list of menu items (similar to a dialog) when the user performs a long-click (press and hold) on a view that declares support for a context menu. Users can perform a contextual action on one item at a time. For example, in the main activity, a list of country names will be listed in listview. On long pressing an item, the contextual menu will be appeared. There will be three menu items in the context menu such as “Edit”, “Share” and “Delete”.



Screenshots of a floating context menu (left) and the contextual action bar (right).

- In the [contextual action mode](#). This mode is a system implementation of [ActionMode](#) that displays a *contextual action bar* at the top of the screen with action items that affect the selected item(s). When this mode is active, users can perform an action on multiple items at once (if your app allows it).

Creating a floating context menu

To provide a floating context menu:

1. Register the [View](#) to which the context menu should be associated by calling [registerForContextMenu\(\)](#) and pass it the [View](#). If your activity uses a [ListView](#) or [GridView](#) and you want each item to provide the same context menu, register all items for a context menu by passing the [ListView](#) or [GridView](#) to [registerForContextMenu\(\)](#).

2. Implement the [onCreateContextMenu\(\)](#) method in your [Activity](#). When the registered view receives a long-click event, the system calls your [onCreateContextMenu\(\)](#) method. This is where you define the menu items, usually by inflating a menu resource. For example:

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
                               ContextMenuInfo menuInfo)
{
    super.onCreateContextMenu(menu, v, menuInfo);
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.context_menu, menu);
}
```

[MenuInflater](#) allows you to inflate the context menu from a [menu resource](#). The callback method parameters include the [View](#) that the user selected and a [ContextMenu, ContextMenuInfo](#) object that provides additional information about the item selected. If your activity has several views that each provides a different context menu, you might use these parameters to determine which context menu to inflate.

3. Implement [onContextItemSelected\(\)](#).

When the user selects a menu item, the system calls this method so you can perform the appropriate action. For example:

```
@Override
public boolean onContextItemSelected(MenuItem item) {
    AdapterContextMenuInfo info = (AdapterContextMenuInfo) item.getContextMenuInfo();
    switch (item.getItemId())
    {
        case R.id.edit:
            editNote(info.id);
            return true;
        case R.id.delete:
            deleteNote(info.id);
            return true;
        default:
            return super.onContextItemSelected(item);
    }
}
```

The [getItemId\(\)](#) method queries the ID for the selected menu item, which you should assign to each menu item in XML using the `android:id` attribute, as shown in the section about [Defining a Menu in XML](#). When you successfully handle a menu item, return `true`. If you don't handle the menu item, you should pass the menu item to the superclass implementation.

Using the contextual action mode

The contextual action mode is a system implementation of [ActionMode](#) that focuses user interaction toward performing contextual actions. When a user enables this mode by selecting an item, a *contextual action bar* appears at the top of the screen to present actions the user can perform on the currently selected item(s). While this mode is enabled, the user can select multiple items (if you allow it), deselect items, and continue to navigate within the activity. The action mode is disabled and the contextual action bar disappears when the user deselects all items, presses the BACK button, or selects the *Done* action on the left side of the bar.

For views that provide contextual actions, you should usually invoke the contextual action mode upon one of two events (or both):

- The user performs a long-click on the view.
- The user selects a checkbox or similar UI component within the view.

How your application invokes the contextual action mode and defines the behaviour for each action depends on your design. There are basically two designs:

- For contextual actions on individual, arbitrary views.
- For batch contextual actions on groups of items in a [ListView](#) or [GridView](#) (allowing the user to select multiple items and perform an action on them all).

Enabling the contextual action mode for individual views

If you want to invoke the contextual action mode only when the user selects specific views, you should:

1. Implement the [ActionMode.Callback](#) interface. In its callback methods, you can specify the actions for the contextual action bar, respond to click events on action items, and handle other lifecycle events for the action mode.
2. Call [startActionMode\(\)](#) when you want to show the bar (such as when the user long-clicks the view).

For example:

1. Implement the [ActionMode.Callback](#) interface:

```
private ActionMode.Callback mActionModeCallback = new ActionMode.Callback() {

    // Called when the action mode is created; startActionMode() was called
    @Override
    public boolean onCreateActionMode(ActionMode mode, Menu menu) {
        // Inflate a menu resource providing context menu items
        MenuInflater inflater = mode.getMenuInflater();
        inflater.inflate(R.menu.context_menu, menu);
        return true;
    }

    // Called each time the action mode is shown. Always called after onCreateActionMode, but
    // may be called multiple times if the mode is invalidated.
    @Override
    public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
        return false; // Return false if nothing is done
    }

    // Called when the user selects a contextual menu item
    @Override
    public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
        switch (item.getItemId()) {
            case R.id.menu_share:
                shareCurrentItem();
                mode.finish(); // Action picked, so close the CAB (Contextual Action Bar)
                return true;
            default:
                return false;
        }
    }
}
```

```

    }
}

// Called when the user exits the action mode
@Override
public void onDestroyActionMode(ActionMode mode) {
    mActionMode = null;
}
};

```

2. Call [startActionMode\(\)](#) to enable the contextual action mode when appropriate, such as in response to a long-click on a [View](#):

```

someView.setOnLongClickListener(new View.OnLongClickListener() {
    // Called when the user long-clicks on someView
    public boolean onLongClick(View view) {
        if (mActionMode != null) {
            return false;
        }

        // Start the CAB using the ActionMode.Callback defined above
        mActionMode = getActivity().startActionMode(mActionModeCallback);
        view.setSelected(true);
        return true;
    }
});

```

When you call [startActionMode\(\)](#), the system returns the [ActionMode](#) created. By saving this in a member variable, you can make changes to the contextual action bar in response to other events. In the above sample, the [ActionMode](#) is used to ensure that the [ActionMode](#) instance is not recreated if it's already active, by checking whether the member is null before starting the action mode.

Enabling batch contextual actions in a [ListView](#) or [GridView](#)

If you have a collection of items in a [ListView](#) or [GridView](#) and want to allow users to perform batch actions, you should:

- Implement the [AbsListView.MultiChoiceModeListener](#) interface and set it for the view group with [setMultiChoiceModeListener\(\)](#). In the listener's callback methods, you can specify the actions for the contextual action bar, respond to click events on action items, and handle other callbacks inherited from the [ActionMode.Callback](#) interface.
- Call [setChoiceMode\(\)](#) with the [CHOICE_MODE_MULTIPLE_MODAL](#) argument.

For example:

```

ListView listView = getListView();
listView.setChoiceMode(ListView.CHOICE_MODE_MULTIPLE_MODAL);
listView.setMultiChoiceModeListener(new MultiChoiceModeListener() {

    @Override
    public void onItemCheckedStateChanged(ActionMode mode, int position,
        long id, boolean checked) {
        // Here you can do something when items are selected/de-selected,
        // such as update the title in the CAB
    }
});

```

```

    }

    @Override
    public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
        // Respond to clicks on the actions in the CAB
        switch (item.getItemId()) {
            case R.id.menu_delete:
                deleteSelectedItems();
                mode.finish(); // Action picked, so close the CAB
                return true;
            default:
                return false;
        }
    }

    @Override
    public boolean onCreateActionMode(ActionMode mode, Menu menu) {
        // Inflate the menu for the CAB
        MenuInflater inflater = mode.getMenuInflater();
        inflater.inflate(R.menu.context, menu);
        return true;
    }

    @Override
    public void onDestroyActionMode(ActionMode mode) {
        // Here you can make any necessary updates to the activity when
        // the CAB is removed. By default, selected items are deselected/unchecked.
    }

    @Override
    public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
        // Here you can perform updates to the CAB due to
        // an invalidate() request
        return false;
    }
}
});

```

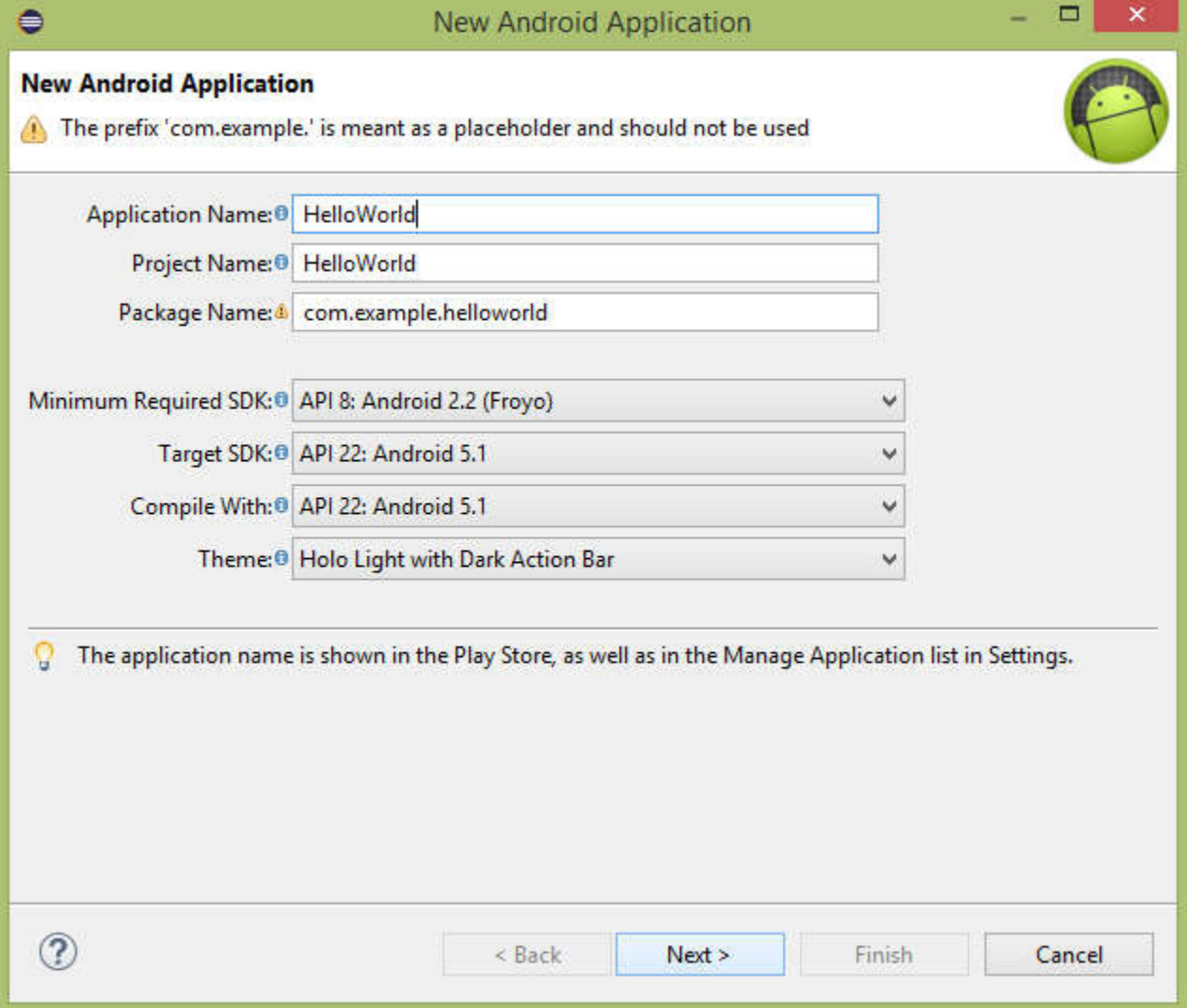
That's it. Now when the user selects an item with a long-click, the system calls the [onCreateActionMode\(\)](#) method and displays the contextual action bar with the specified actions. While the contextual action bar is visible, users can select additional items.

In some cases in which the contextual actions provide common action items, you might want to add a checkbox or a similar UI element that allows users to select items, because they might not discover the long-click behavior. When a user selects the checkbox, you can invoke the contextual action mode by setting the respective list item to the checked state with [setItemChecked\(\)](#).

Android - Hello World Example

Create Android Application

The first step is to create a simple Android Application using Eclipse IDE. Follow the option **File -> New -> Project** and finally select **Android New Application** wizard from the wizard list. Now name your application as **HelloWorld** using the wizard window as follows:



The screenshot shows the 'New Android Application' wizard window. The title bar is green with the text 'New Android Application'. The main content area has a light gray background. At the top, there is a warning icon and the text: 'The prefix 'com.example.' is meant as a placeholder and should not be used'. Below this, there are several input fields and dropdown menus. The 'Application Name' field contains 'HelloWorld'. The 'Project Name' field contains 'HelloWorld'. The 'Package Name' field contains 'com.example.helloworld'. The 'Minimum Required SDK' dropdown is set to 'API 8: Android 2.2 (Froyo)'. The 'Target SDK' dropdown is set to 'API 22: Android 5.1'. The 'Compile With' dropdown is set to 'API 22: Android 5.1'. The 'Theme' dropdown is set to 'Holo Light with Dark Action Bar'. At the bottom, there is a light bulb icon and the text: 'The application name is shown in the Play Store, as well as in the Manage Application list in Settings.' The bottom of the window has a navigation bar with a question mark icon, a '< Back' button, a 'Next >' button (which is highlighted in blue), a 'Finish' button, and a 'Cancel' button.

New Android Application

⚠ The prefix 'com.example.' is meant as a placeholder and should not be used

Application Name: HelloWorld

Project Name: HelloWorld

Package Name: com.example.helloworld

Minimum Required SDK: API 8: Android 2.2 (Froyo)

Target SDK: API 22: Android 5.1

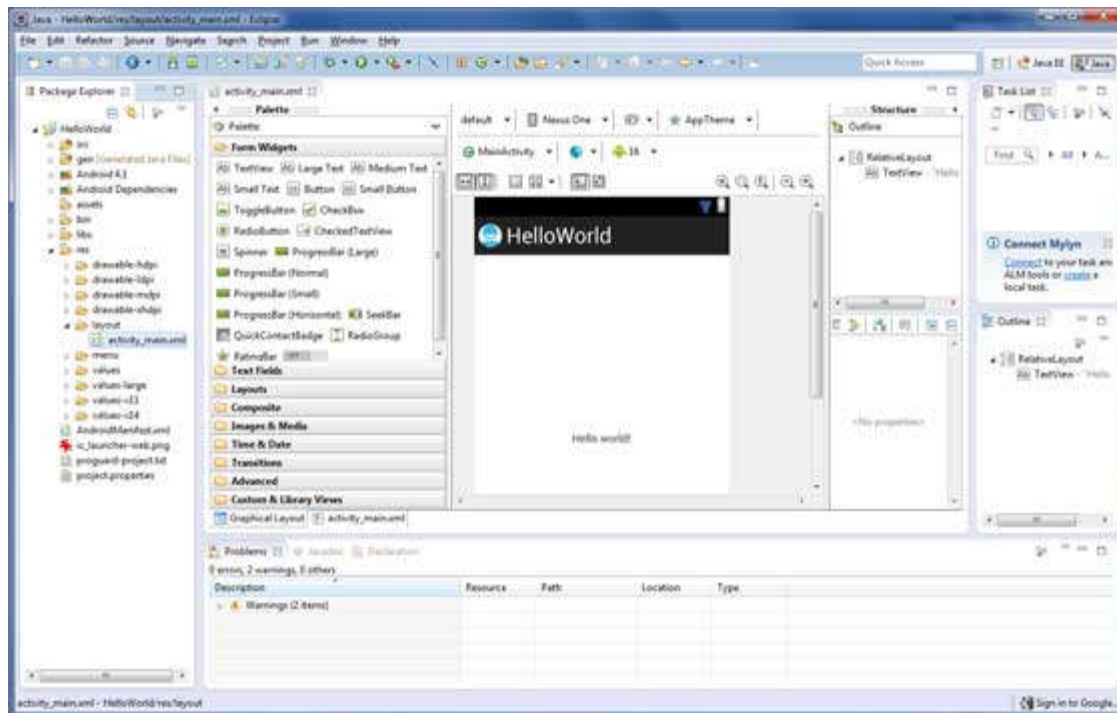
Compile With: API 22: Android 5.1

Theme: Holo Light with Dark Action Bar

💡 The application name is shown in the Play Store, as well as in the Manage Application list in Settings.

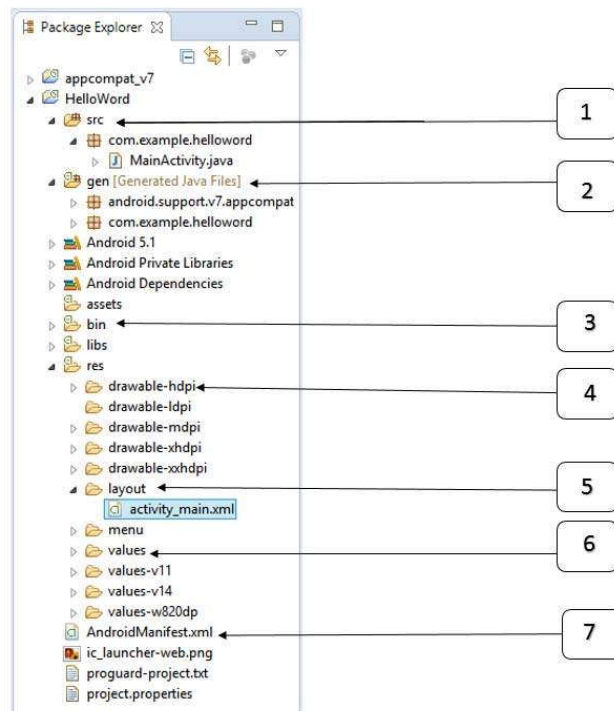
? < Back Next > Finish Cancel

Next, follow the instructions provided and keep all other entries as default till the final step. Once your project is created successfully, you will have following project screen –



Anatomy of Android Application

Before you run your app, you should be aware of a few directories and files in the Android project –



S.N.	Folder, File & Description
------	----------------------------

src

- | | |
|---|--|
| 1 | This contains the .java source files for your project. By default, it includes an <i>MainActivity.java</i> source file having an activity class that runs when your app is launched using the app icon. |
|---|--|

gen

- | | |
|---|---|
| 2 | This contains the .R file, a compiler-generated file that references all the resources found in your project. You should not modify this file. |
|---|---|

bin

- | | |
|---|--|
| 3 | This folder contains the Android package files .apk built by the ADT during the build process and everything else needed to run an Android application. |
|---|--|

res/drawable-hdpi

- | | |
|---|--|
| 4 | This is a directory for drawable objects that are designed for high-density screens. |
|---|--|

res/layout

- | | |
|---|--|
| 5 | This is a directory for files that define your app's user interface. |
|---|--|

res/values

- | | |
|---|--|
| 6 | This is a directory for other various XML files that contain a collection of resources, such as strings and colours definitions. |
|---|--|

AndroidManifest.xml

- | | |
|---|--|
| 7 | This is the manifest file which describes the fundamental characteristics of the app and defines each of its components. |
|---|--|

Following section will give a brief overview few of the important application files.

The Main Activity File

The main activity code is a Java file **MainActivity.java**. This is the actual application file which ultimately gets converted to a Dalvik executable and runs your application. Following is the default code generated by the application wizard for *Hello World!* application –

```
package com.example.helloworld;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.view.MenuItem;
import android.support.v4.app.NavUtils;

public class MainActivity extends Activity {

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

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.activity_main, menu);
        return true;
    }
}
```

Here, *R.layout.activity_main* refers to the *activity_main.xml* file located in the *res/layout* folder. The *onCreate()* method is one of many methods that are figured when an activity is loaded.

The Manifest File

Whatever component you develop as a part of your application, you must declare all its components in a *manifest.xml* which resides at the root of the application project directory. This file works as an interface between Android OS and your application, so if you do not declare your component in this file, then it will not be considered by the OS. For example, a default manifest file will look like as following file –

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.helloworld"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="22" />

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

        <activity
            android:name=".MainActivity"
            android:label="@string/title_activity_main" >
```

```

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

    </activity>
</application>
</manifest>

```

Here `<application>...</application>` tags enclosed the components related to the application. Attribute `android:icon` will point to the application icon available under `res/drawable-hdpi`. The application uses the image named `ic_launcher.png` located in the drawable folders

The `<activity>` tag is used to specify an activity and `android:name` attribute specifies the fully qualified class name of the *Activity* subclass and the `android:label` attributes specifies a string to use as the label for the activity. You can specify multiple activities using `<activity>` tags.

The **action** for the intent filter is named `android.intent.action.MAIN` to indicate that this activity serves as the entry point for the application. The **category** for the intent-filter is named `android.intent.category.LAUNCHER` to indicate that the application can be launched from the device's launcher icon.

The `@string` refers to the `strings.xml` file explained below. Hence, `@string/app_name` refers to the `app_name` string defined in the strings.xml file, which is "HelloWorld". Similar way, other strings get populated in the application.

Following is the list of tags which you will use in your manifest file to specify different Android application components:

- `<activity>` elements for activities
- `<service>` elements for services
- `<receiver>` elements for broadcast receivers
- `<provider>` elements for content providers

The Strings File

The **strings.xml** file is located in the `res/values` folder and it contains all the text that your application uses. For example, the names of buttons, labels, default text, and similar types of strings go into this file. This file is responsible for their textual content. For example, a default strings file will look like as following file –

```

<resources>
    <string name="app_name">HelloWorld</string>
    <string name="hello_world">Hello world!</string>
    <string name="menu_settings">Settings</string>
    <string name="title_activity_main">MainActivity</string>
</resources>

```

The R File

The **gen/com.example.helloworld/R.java** file is the glue between the activity Java files like `MainActivity.java` and the resources like `strings.xml`. It is an automatically generated file and you should not modify the content of the R.java file. Following is a sample of R.java file –

```

/* AUTO-GENERATED FILE. DO NOT MODIFY.
 *
 * This class was automatically generated by the aapt tool from the resource data it found. It should not be
 * modified by hand.
 */
package com.example.helloworld;

```

```

public final class R {
    public static final class attr {
    }

    public static final class dimen {
        public static final int padding_large=0x7f040002;
        public static final int padding_medium=0x7f040001;
        public static final int padding_small=0x7f040000;
    }

    public static final class drawable {
        public static final int ic_action_search=0x7f020000;
        public static final int ic_launcher=0x7f020001;
    }

    public static final class id {
        public static final int menu_settings=0x7f080000;
    }

    public static final class layout {
        public static final int activity_main=0x7f030000;
    }

    public static final class menu {
        public static final int activity_main=0x7f070000;
    }

    public static final class string {
        public static final int app_name=0x7f050000;
        public static final int hello_world=0x7f050001;
        public static final int menu_settings=0x7f050002;
        public static final int title_activity_main=0x7f050003;
    }

    public static final class style {
        public static final int AppTheme=0x7f060000;
    }
}

```

R is a class that contains only public constants. It is generated class that reflects the various values you defined in the res file. It contains assigned numeric constant for each resource that you can refer in your project. For example, you have XML resource file that contains `about_button`. If you didn't have R class, you would have to use a string "about_button" to refer to it in code. If you make a mistake in this string, you will only learn about it when you run your application. With R you will see the error much earlier at compile time.

R is structured in such a way that you can refer to resources via its inner classes. For example, `R.id` contains id constants and `R.layout` contains layout constants.

The Layout File

The **activity_main.xml** is a layout file available in `res/layout` directory, which is referenced by application when building its interface. You will modify this file very frequently to change the layout of your application. For your "Hello World!" application, this file will have following content related to default layout –

```

<RelativeLayout 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" >

    <TextView


```

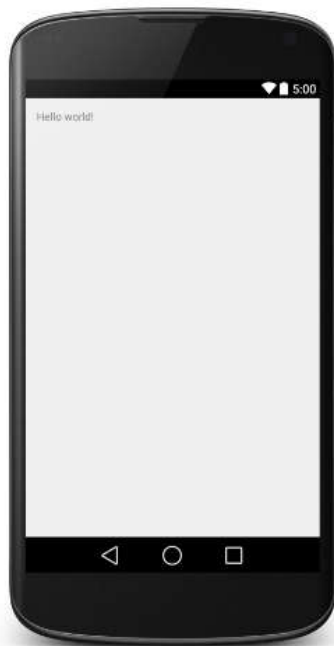
```
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_centerHorizontal="true"
android:layout_centerVertical="true"
android:padding="@dimen/padding_medium"
android:text="@string/hello_world"
tools:context=".MainActivity" />
```

</RelativeLayout>

This is an example of simple *RelativeLayout* which we will study in a separate chapter. The *TextView* is an Android control used to build the GUI and it have various attributes like *android:layout_width*, *android:layout_height* etc which are being used to set its width and height etc.. The *@string* refers to the strings.xml file located in the res/values folder. Hence, *@string/hello_world* refers to the hello string defined in the strings.xml file, which is "Hello World!".

Running the Application

Let's try to run our **Hello World!** application we just created. I assume you had created your **AVD** while doing environment set-up. To run the app from Eclipse, open one of your project's activity files and click Run  icon from the tool bar. Eclipse installs the app on your AVD and starts it and if everything is fine with your set-up and application, it will display following Emulator window –



Congratulations!!! you have developed your first Android Application and now just keep following rest of the tutorial step by step to become a great Android Developer. All the very best.

Create Android Application on Android studio

There are so many tools are available to develop android applications. as per Google official android development, they have launched android studio as official Android IDE