ANDROID STUDIO

INTRODUCTION

***Android Studio (Basics)***

**What is Android?**



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

Android offers a unified approach to application development for mobile devices which means developers need only develop for Android, and their applications should be able to run on different devices powered by Android.

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.

**Why Android ?**

**Features of Android**

Android is a powerful operating system competing with Apple 4GS and supports great features. Few of them are listed below −

|  |  |
| --- | --- |
| **Sr.No.** | **Feature & Description** |
| 1 | **Beautiful UI**  Android OS basic screen provides a beautiful and intuitive user interface. |
| 2 | **Connectivity**  GSM/EDGE, IDEN, CDMA, EV-DO, UMTS, Bluetooth, Wi-Fi, LTE, NFC and WiMAX. |
| 3 | **Storage**  SQLite, a lightweight relational database, is used for data storage purposes. |
| 4 | **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. |
| 5 | **Messaging**  SMS and MMS |
| 6 | **Web browser**  Based on the open-source WebKit layout engine, coupled with Chrome's V8 JavaScript engine supporting HTML5 and CSS3. |
| 7 | **Multi-touch**  Android has native support for multi-touch which was initially made available in handsets such as the HTC Hero. |
| 8 | **Multi-tasking**  User can jump from one task to another and same time various application can run simultaneously. |
| 9 | **Resizable widgets**  Widgets are resizable, so users can expand them to show more content or shrink them to save space. |
| 10 | **Multi-Language**  Supports single direction and bi-directional text. |
| 11 | **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. |
| 12 | **Wi-Fi Direct**  A technology that lets apps discover and pair directly, over a high-bandwidth peer-to-peer connection. |
| 13 | **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 Appstore**.

Android powers hundreds of millions of mobile devices in more than 190 countries around the world. It's the largest installed base of any mobile platform and growing fast. Every day more than 1 million new Android devices are activated worldwide.

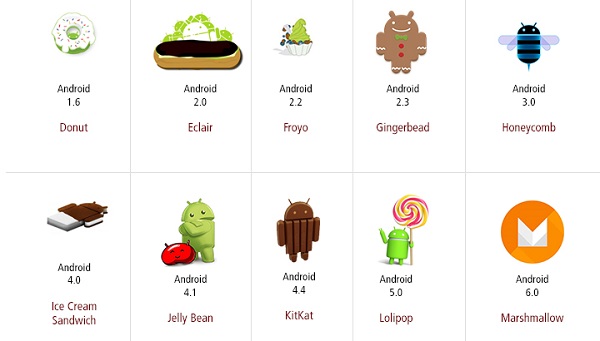
This tutorial has been written with an aim to teach you how to develop and package Android application. We will start from environment setup for Android application programming and then drill down to look into various aspects of Android applications.

**Categories of Android applications**

There are many android applications in the market. The top categories are −

**History of Android**

The code names of android ranges from A to N currently, such as Aestro, Blender, Cupcake, Donut, Eclair, Froyo, Gingerbread, Honeycomb, Ice Cream Sandwitch, Jelly Bean, KitKat, Lollipop and Marshmallow. Let's understand the android history in a sequence.



**What is API level?**

API Level is an integer value that uniquely identifies the framework API revision offered by a version of the Android platform.

|  |  |  |  |
| --- | --- | --- | --- |
| **Platform Version** | **API Level** | **VERSION\_CODE** |  |
| Android 6.0 | 23 | MARSHMALLOW |  |
| Android 5.1 | 22 | LOLLIPOP\_MR1 |  |
| Android 5.0 | 21 | LOLLIPOP |  |
| Android 4.4W | 20 | KITKAT\_WATCH | KitKat for Wearables Only |
| Android 4.4 | 19 | KITKAT |  |
| Android 4.3 | 18 | JELLY\_BEAN\_MR2 |  |
| Android 4.2, 4.2.2 | 17 | JELLY\_BEAN\_MR1 |  |
| Android 4.1, 4.1.1 | 16 | JELLY\_BEAN |  |
| Android 4.0.3, 4.0.4 | 15 | ICE\_CREAM\_SANDWICH\_MR1 |  |
| Android 4.0, 4.0.1, 4.0.2 | 14 | ICE\_CREAM\_SANDWICH |  |
| Android 3.2 | 13 | HONEYCOMB\_MR2 |  |
| Android 3.1.x | 12 | HONEYCOMB\_MR1 |  |
| Android 3.0.x | 11 | HONEYCOMB |  |
| Android 2.3.4  Android 2.3.3 | 10 | GINGERBREAD\_MR1 |  |
| Android 2.3.2  Android 2.3.1  Android 2.3 | 9 | GINGERBREAD |  |
| Android 2.2.x | 8 | FROYO |  |
| Android 2.1.x | 7 | ECLAIR\_MR1 |  |
| Android 2.0.1 | 6 | ECLAIR\_0\_1 |  |
| Android 2.0 | 5 | ECLAIR |  |
| Android 1.6 | 4 | DONUT |  |
| Android 1.5 | 3 | CUPCAKE |  |
| Android 1.1 | 2 | BASE\_1\_1 |  |
| Android 1.0 | 1 | BASE |  |

**ENVIRONMENT SETTINGS**



You will be glad to know that you can start your Android application development on either of the following operating systems −

* Microsoft Windows XP or later version.
* Mac OS X 10.5.8 or later version with Intel chip.
* Linux including GNU C Library 2.7 or later.

Second point is that all the required tools to develop Android applications are freely available and can be downloaded from the Web. Following is the list of software's you will need before you start your Android application programming.

* Java JDK5 or later version
* Android Studio

Here last two components are optional and if you are working on Windows machine then these components make your life easy while doing Java based application development. So let us have a look how to proceed to set required environment.

**Set-up Java Development Kit (JDK)**

You can download the latest version of Java JDK from Oracle's Java site − [Java SE Downloads](http://www.oracle.com/technetwork/java/javase/downloads/index.html). You will find instructions for installing JDK in downloaded files, follow the given instructions to install and configure the setup. Finally set PATH and JAVA\_HOME environment variables to refer to the directory that contains **java** and **javac**, typically java\_install\_dir/bin and java\_install\_dir respectively.

If you are running Windows and installed the JDK in C:\jdk1.8.0\_102, you would have to put the following line in your C:\autoexec.bat file.

set PATH=C:\jdk1.8.0\_102\bin;%PATH%

set JAVA\_HOME=C:\jdk1.8.0\_102

Alternatively, you could also right-click on *My Computer*, select *Properties*, then *Advanced*, then *Environment Variables*. Then, you would update the PATH value and press the OK button.

On Linux, if the SDK is installed in /usr/local/jdk1.8.0\_102 and you use the C shell, you would put the following code into your **.cshrc** file.

setenv PATH /usr/local/jdk1.8.0\_102/bin:$PATH

setenv JAVA\_HOME /usr/local/jdk1.8.0\_102

Alternatively, if you use Android studio, then it will know automatically where you have installed your Java.

**Android IDEs**

There are so many sophisticated Technologies are available to develop android applications, the familiar technologies, which are predominantly using tools as follows

* [Android Studio](https://www.tutorialspoint.com/android/android_studio.htm)
* [Eclipse IDE(Deprecated)](https://www.tutorialspoint.com/android/android_eclipse.htm)

# Android – Architecture

Android operating system is a stack of software components which is roughly divided into five sections and four main layers as shown below in the architecture diagram.



## Linux kernel

At the bottom of the layers is Linux - Linux 3.6 with approximately 115 patches. This provides a level of abstraction between the device hardware and it contains all the essential hardware drivers like camera, keypad, display etc. Also, the kernel handles all the things that Linux is really good at such as networking and a vast array of device drivers, which take the pain out of interfacing to peripheral hardware.

## Libraries

On top of Linux kernel there is a set of libraries including open-source Web browser engine WebKit, well known library libc, 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 Libraries

This category encompasses those Java-based libraries that are specific to Android development. Examples of libraries in this category include the application framework libraries in addition to those that facilitate user interface building, graphics drawing and database access. A summary of some key core Android libraries available to the Android developer is as follows −

* **android.app** − Provides access to the application model and is the cornerstone of all Android applications.
* **android.content** − Facilitates content access, publishing and messaging between applications and application components.
* **android.database** − Used to access data published by content providers and includes SQLite database management classes.
* **android.opengl** − A Java interface to the OpenGL ES 3D graphics rendering API.
* **android.os** − Provides applications with access to standard operating system services including messages, system services and inter-process communication.
* **android.text** − Used to render and manipulate text on a device display.
* **android.view** − The fundamental building blocks of application user interfaces.
* **android.widget** − A rich collection of pre-built user interface components such as buttons, labels, list views, layout managers, radio buttons etc.
* **android.webkit** − A set of classes intended to allow web-browsing capabilities to be built into applications.

Having covered the Java-based core libraries in the Android runtime, it is now time to turn our attention to the C/C++ based libraries contained in this layer of the Android software stack.

## Android Runtime

This is the third section of the architecture and available on the second layer from the bottom. This section provides a key component called **Dalvik Virtual Machine** which is a kind of Java Virtual Machine specially designed and optimized for Android.

The Dalvik VM makes use of Linux core features like memory management and multi-threading, which is intrinsic in the Java language. The Dalvik VM enables every Android application to run in its own process, with its own instance of the Dalvik virtual machine.

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

## Application Framework

The Application Framework layer provides many higher-level services to applications in the form of Java classes. Application developers are allowed to make use of these services in their applications.

The Android framework includes the following key services −

* **Activity Manager** − Controls all aspects of the application lifecycle and activity stack.
* **Content Providers** − Allows applications to publish and share data with other applications.
* **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

You will find all the Android application at the top layer. You will write your application to be installed on this layer only. Examples of such applications are Contacts Books, Browser, Games etc.

# Android - Application Components

Application components are the essential building blocks of an Android application. These components are loosely coupled by the application manifest file *AndroidManifest.xml* that describes each component of the application and how they interact.

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

|  |  |
| --- | --- |
| **Sr.No** | **Components & Description** |
| 1 | **Activities**  They dictate the UI and handle the user interaction to the smart phone screen. |
| 2 | **Services**  They handle background processing associated with an application. |
| 3 | **Broadcast Receivers**  They handle communication between Android OS and applications. |
| 4 | **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, 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.

An activity is implemented as a subclass of **Activity** class as follows −

public class MainActivity extends Activity {

}

**Services**

A service is a component that runs in the background to perform long-running operations. For example, a service might play music in the background while the user is in a different application, or it might fetch data over the network without blocking user interaction with an activity.

A service is implemented as a subclass of **Service** class as follows −

public class MyService extends Service {

}

**Broadcast Receivers**

Broadcast Receivers simply respond to broadcast messages from other applications or from the system. 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.

A broadcast receiver is implemented as a subclass of **BroadcastReceiver** class and each message is broadcaster as an **Intent** object.

public class MyReceiver extends BroadcastReceiver {

public void onReceive(context,intent){}

}

**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. The data may be stored in the file system, the database or somewhere else entirely.

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 MyContentProvider extends ContentProvider {

public void onCreate(){}

}

We will go through these tags in detail while covering application components in individual chapters.

**Additional Components**

There are additional components which will be used in the construction of above mentioned entities, their logic, and wiring between them. These components are −

|  |  |
| --- | --- |
| **S.No** | **Components & Description** |
| 1 | **Fragments**  Represents a portion of user interface in an Activity. |
| 2 | **Views**  UI elements that are drawn on-screen including buttons, lists forms etc. |
| 3 | **Layouts**  View hierarchies that control screen format and appearance of the views. |
| 4 | **Intents**  Messages wiring components together. |
| 5 | **Resources**  External elements, such as strings, constants and drawable pictures. |
| 6 | **Manifest**  Configuration file for the application. |

# Android - Hello World Example

Let us start actual programming with Android Framework. Before you start writing your first example using Android SDK, you have to make sure that you have set-up your Android development environment properly as explained in [Android - Environment Set-up](https://www.tutorialspoint.com/android/android_environment_setup.htm) tutorial. I also assume that you have a little bit working knowledge with Android studio.

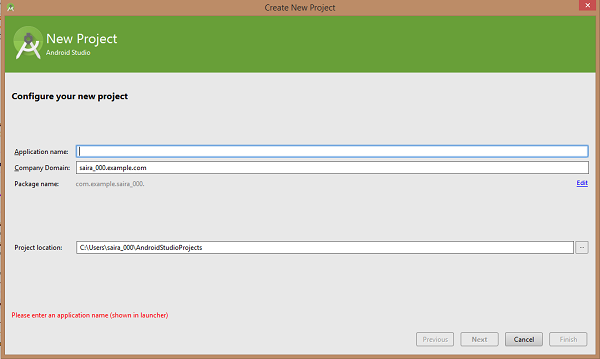
So let us proceed to write a simple Android Application which will print "Hello World!".

## Create Android Application

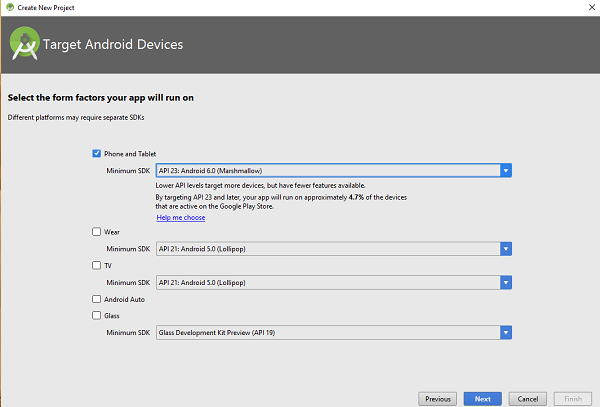
The first step is to create a simple Android Application using Android studio. When you click on Android studio icon, it will show screen as shown below



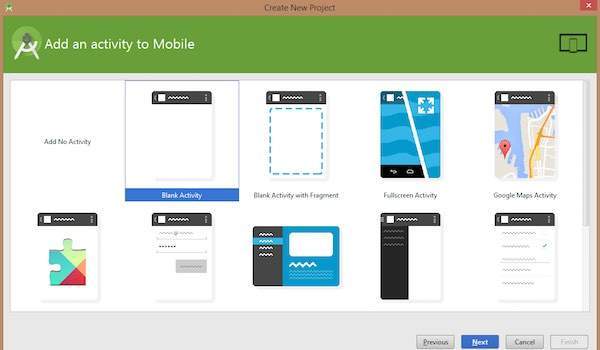
You can start your application development by calling start a new android studio project. in a new installation frame should ask Application name, package information and location of the project.−



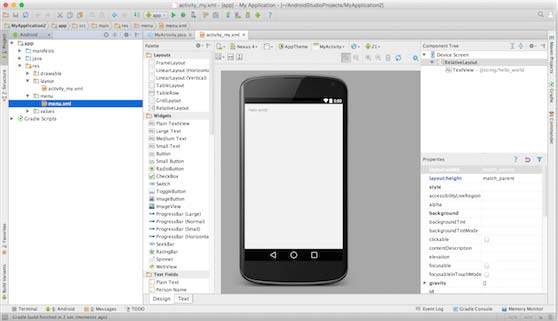
After entered application name, it going to be called select the form factors your application runs on, here need to specify Minimum SDK, in our tutorial, I have declared as API23: Android 6.0(Mashmallow) −



The next level of installation should contain selecting the activity to mobile, it specifies the default layout for Applications.

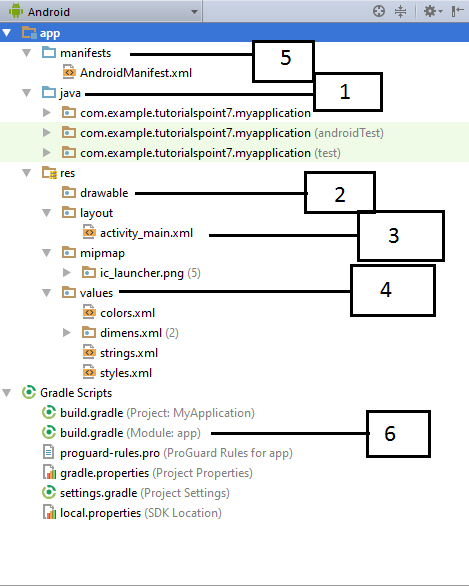


At the final stage it going to be open development tool to write the application code.



**Anatomy of Android Application**

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



|  |  |
| --- | --- |
| **Sr.No.** | **Folder, File & Description** |
| 1 | **Java**  This contains the **.java** source files for your project. By default, it includes an *MainActivity.java* source file having an activity class that runs when your app is launched using the app icon. |
|  |  |
| 2 | **res/drawable-hdpi**  This is a directory for drawable objects that are designed for high-density screens. |
| 3 | **res/layout**  This is a directory for files that define your app's user interface. |
| 4 | **res/values**  This is a directory for other various XML files that contain a collection of resources, such as strings and colours definitions. |
| 5 | **AndroidManifest.xml**  This is the manifest file which describes the fundamental characteristics of the app and defines each of its components. |
| 6 | **Build.gradle**  This is an auto generated file which contains compileSdkVersion, buildToolsVersion, applicationId, minSdkVersion, targetSdkVersion, versionCode and versionName |

Following section will give a brief overview 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.support.v7.app.AppCompatActivity;

import android.os.Bundle;

public class MainActivity extends AppCompatActivity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

}

}

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 −

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

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.tutorialspoint7.myapplication">

<application

android:allowBackup="true"

android:icon="@mipmap/ic\_launcher"

android:label="@string/app\_name"

android:supportsRtl="true"

android:theme="@style/AppTheme">

<activity android:name=".MainActivity">

<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 Layout File**

The **activity\_main.xml** is a layout file available in *res/layout* directory, that is referenced by your 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 Android studio, open one of your project's activity files and click Run Eclipse Run Iconicon from the tool bar. Android studio 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.

# Android Resources Organizing & Accessing

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.

This tutorial will explain you how you can organize your application resources, specify alternative resources and access them in your applications.

## Organize resource in Android Studio

MyProject/

app/

manifest/

AndroidManifest.xml

java/

MyActivity.java

**res/**

drawable/

icon.png

layout/

activity\_main.xml

info.xml

values/

strings.xml

|  |  |
| --- | --- |
| **Sr.No.** | **Directory & Resource Type** |
| 1 | **anim/**  XML files that define property animations. They are saved in res/anim/ folder and accessed from the **R.anim** class. |
| 2 | **color/**  XML files that define a state list of colors. They are saved in res/color/ and accessed from the **R.color** class. |
| 3 | **drawable/**  Image files like .png, .jpg, .gif or XML files that are compiled into bitmaps, state lists, shapes, animation drawable. They are saved in res/drawable/ and accessed from the **R.drawable** class. |
| 4 | **layout/**  XML files that define a user interface layout. They are saved in res/layout/ and accessed from the **R.layout** class. |
| 5 | **menu/**  XML files that define application menus, such as an Options Menu, Context Menu, or Sub Menu. They are saved in res/menu/ and accessed from the **R.menu** class. |
| 6 | **raw/**  Arbitrary files to save in their raw form. You need to call *Resources.openRawResource()* with the resource ID, which is *R.raw.filename* to open such raw files. |
| 7 | **values/**  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 −   * arrays.xml for resource arrays, and accessed from the **R.array** class. * integers.xml for resource integers, and accessed from the **R.integer** class. * bools.xml for resource boolean, and accessed from the **R.bool** class. * colors.xml for color values, and accessed from the **R.color** class. * dimens.xml for dimension values, and accessed from the **R.dimen** class. * strings.xml for string values, and accessed from the **R.string** class. * styles.xml for styles, and accessed from the **R.style** class. |
| 8 | **xml/**  Arbitrary XML files that can be read at runtime by calling *Resources.getXML()*. You can save various configuration files here which will be used at run time. |

## 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

Below is another example which specifies layout for a default language and alternative layout for Arabic language.

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

**layout-ar/**

main.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 your 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"**

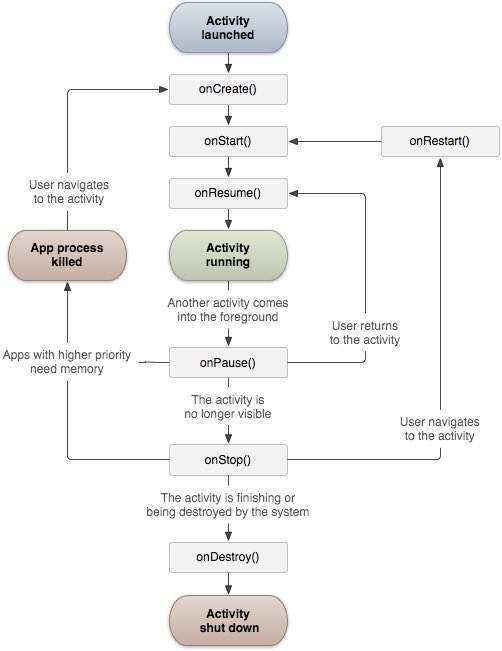
android:text=**"@string/hello" />**

Now if you will go through previous chapter once again where I have explained **Hello World!** example, and I'm sure you will have better understanding on all the concepts explained in this chapter. So I highly recommend to check previous chapter for working example and check how I have used various resources at very basic level.

# Android - Activities

An activity represents a single screen with a user interface just like window or frame of Java.Android activity is the subclass of ContextThemeWrapper class.

If you have worked with C, C++ or Java programming language then you must have seen that your program starts from **main()** function. Very similar way, Android system initiates its program with in an **Activity** starting with a call on *onCreate()* callback method. There is a sequence of callback methods that start up an activity and a sequence of callback methods that tear down an activity as shown in the below Activity life cycle diagram: (*image courtesy : android.com* )



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 | **onCreate()**  This is the first callback and called when the activity is first created. |
| 2 | **onStart()**  This callback is called when the activity becomes visible to the user. |
| 3 | **onResume()**  This is called when the user starts interacting with the application. |
| 4 | **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. |
| 5 | **onStop()**  This callback is called when the activity is no longer visible. |
| 6 | **onDestroy()**  This callback is called before the activity is destroyed by the system. |
| 7 | **onRestart()**  This callback is called when the activity restarts after stopping it. |

**Example**

This example will take you through simple steps to show Android application activity life cycle. Follow the following steps to modify the Android application we created in *Hello World Example* chapter −

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use Android studio to create an Android application and name it as *HelloWorld* under a package *com.example.helloworld* as explained in the *Hello World Example* chapter. |
| 2 | Modify main activity file *MainActivity.java* as explained below. Keep rest of the files unchanged. |
| 3 | 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.helloworld/MainActivity.java**. This file includes each of the fundamental life cycle methods. The **Log.d()** method has been used to generate log messages −

package com.example.helloworld;

import android.os.Bundle;

import android.app.Activity;

import android.util.Log;

public class MainActivity extends Activity {

String msg = "Android : ";

/\*\* Called when the activity is first created. \*/

@Override

public void **onCreate**(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

Log.d(msg, "The onCreate() event");

}

/\*\* Called when the activity is about to become visible. \*/

@Override

protected void **onStart()** {

super.onStart();

Log.d(msg, "The onStart() event");

}

/\*\* Called when the activity has become visible. \*/

@Override

protected void **onResume()** {

super.onResume();

Log.d(msg, "The onResume() event");

}

/\*\* Called when another activity is taking focus. \*/

@Override

protected void **onPause()** {

super.onPause();

Log.d(msg, "The onPause() event");

}

/\*\* Called when the activity is no longer visible. \*/

@Override

protected void **onStop()** {

super.onStop();

Log.d(msg, "The onStop() event");

}

/\*\* Called just before the activity is destroyed. \*/

@Override

public void **onDestroy()** {

super.onDestroy();

Log.d(msg, "The onDestroy() event");

}

}

An activity class loads all the UI component using the XML file available in *res/layout* folder of the project. Following statement loads UI components from *res/layout/activity\_main.xml file*:

setContentView(R.layout.activity\_main);

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> that includes the MAIN action and LAUNCHER category as follows:

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

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.tutorialspoint7.myapplication">

<application

android:allowBackup="true"

android:icon="@mipmap/ic\_launcher"

android:label="@string/app\_name"

android:supportsRtl="true"

android:theme="@style/AppTheme">

<activity android:name=".MainActivity">

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

</application>

</manifest>

If either the MAIN action or LAUNCHER category are not declared for one of your activities, then your app icon will not appear in the Home screen's list of apps.

Let's try to run our modified **Hello World!** application we just modified. I assume you had created your **AVD** while doing environment setup. To run the app from Android studio, open one of your project's activity files and click Run Eclipse Run Iconicon from the toolbar. Android studio installs the app on your AVD and starts it and if everything is fine with your setup and application, it will display Emulator window and you should see following log messages in **LogCat** window in Android studio −

08-23 10:32:07.682 4480-4480/com.example.helloworld D/Android :: The onCreate() event

08-23 10:32:07.683 4480-4480/com.example.helloworld D/Android :: The onStart() event

08-23 10:32:07.685 4480-4480/com.example.helloworld D/Android :: The onResume() event



Let us try to click lock screen button on the Android emulator and it will generate following events messages in **LogCat** window in android studio:

08-23 10:32:53.230 4480-4480/com.example.helloworld D/Android :: The onPause() event

08-23 10:32:53.294 4480-4480/com.example.helloworld D/Android :: The onStop() event

Let us again try to unlock your screen on the Android emulator and it will generate following events messages in **LogCat** window in Android studio:

08-23 10:34:41.390 4480-4480/com.example.helloworld D/Android :: The onStart() event

08-23 10:34:41.392 4480-4480/com.example.helloworld D/Android :: The onResume() event

Next, let us again try to click Back button Android Back Buttonon the Android emulator and it will generate following events messages in **LogCat** window in Android studio and this completes the Activity Life Cycle for an Android Application.

08-23 10:37:24.806 4480-4480/com.example.helloworld D/Android :: The onPause() event

08-23 10:37:25.668 4480-4480/com.example.helloworld D/Android :: The onStop() event

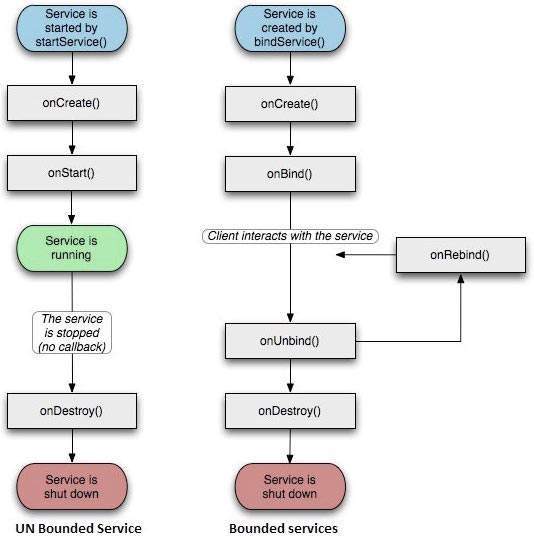
08-23 10:37:25.669 4480-4480/com.example.helloworld D/Android :: The onDestroy() event

# 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 −

|  |  |
| --- | --- |
| **Sr.No.** | **State & Description** |
| 1 | **Started**  A service is **started** when an application component, such as an activity, starts it by calling *startService()*. Once started, a service can run in the background indefinitely, even if the component that started it is destroyed. |
| 2 | **Bound**  A service is **bound** when an application component binds to it by calling *bindService()*. 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). |

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 )*



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.

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

The following skeleton service demonstrates each of the life cycle methods −

package com.tutorialspoint;

import android.app.Service;

import android.os.IBinder;

import android.content.Intent;

import android.os.Bundle;

public class HelloService extends Service {

/\*\* indicates how to behave if the service is killed \*/

int mStartMode;

/\*\* interface for clients that bind \*/

IBinder mBinder;

/\*\* indicates whether onRebind should be used \*/

boolean mAllowRebind;

/\*\* Called when the service is being created. \*/

@Override

public void onCreate() {

}

/\*\* The service is starting, due to a call to startService() \*/

@Override

public int onStartCommand(Intent intent, int flags, int startId) {

return mStartMode;

}

/\*\* A client is binding to the service with bindService() \*/

@Override

public IBinder onBind(Intent intent) {

return mBinder;

}

/\*\* Called when all clients have unbound with unbindService() \*/

@Override

public boolean onUnbind(Intent intent) {

return mAllowRebind;

}

/\*\* Called when a client is binding to the service with bindService()\*/

@Override

public void onRebind(Intent intent) {

}

/\*\* Called when The service is no longer used and is being destroyed \*/

@Override

public void onDestroy() {

}

}

**Example**

This example will take you through simple steps to show how to create your own Android Service. Follow the following steps to modify the Android application we created in *Hello World Example* chapter −

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use Android StudioIDE to create an Android application and name it as *My Application* under a package *com.example.tutorialspoint7.myapplication* as explained in the *Hello World Example* chapter. |
| 2 | Modify main activity file *MainActivity.java* to add *startService()* and *stopService()* methods. |
| 3 | Create a new java file *MyService.java* under the package *com.example.My Application*. This file will have implementation of Android service related methods. |
| 4 | Define your service in *AndroidManifest.xml* file using <service.../> tag. An application can have one or more services without any restrictions. |
| 5 | Modify the default content of *res/layout/activity\_main.xml* file to include two buttons in linear layout. |
| 6 | No need to change any constants in *res/values/strings.xml* file. Android studio take care of string values |
| 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 **MainActivity.java**. This file can include each of the fundamental life cycle methods. We have added *startService()* and *stopService()* methods to start and stop the service.

package com.example.tutorialspoint7.myapplication;

import android.content.Intent;

import android.support.v7.app.AppCompatActivity;

import android.os.Bundle;

import android.os.Bundle;

import android.app.Activity;

import android.util.Log;

import android.view.View;

public class MainActivity extends Activity {

String msg = "Android : ";

/\*\* Called when the activity is first created. \*/

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

Log.d(msg, "The onCreate() event");

}

public void startService(View view) {

startService(new Intent(getBaseContext(), MyService.class));

}

// Method to stop the service

public void stopService(View view) {

stopService(new Intent(getBaseContext(), MyService.class));

}

}

Following is the content of **MyService.java**. This file can have implementation of one or more methods associated with Service based on requirements. For now we are going to implement only two methods *onStartCommand()* and *onDestroy()* −

package com.example.tutorialspoint7.myapplication;

import android.app.Service;

import android.content.Intent;

import android.os.IBinder;

import android.support.annotation.Nullable;

import android.widget.Toast;

/\*\*

\* Created by TutorialsPoint7 on 8/23/2016.

\*/

public class MyService extends Service {

@Nullable

@Override

public IBinder onBind(Intent intent) {

return null;

}

@Override

public int onStartCommand(Intent intent, int flags, int startId) {

// Let it continue running until it is stopped.

Toast.makeText(this, "Service Started", Toast.LENGTH\_LONG).show();

return START\_STICKY;

}

@Override

public void onDestroy() {

super.onDestroy();

Toast.makeText(this, "Service Destroyed", Toast.LENGTH\_LONG).show();

}

}

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

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

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.tutorialspoint7.myapplication">

<application

android:allowBackup="true"

android:icon="@mipmap/ic\_launcher"

android:label="@string/app\_name"

android:supportsRtl="true"

android:theme="@style/AppTheme">

<activity android:name=".MainActivity">

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

<service android:name=".MyService" />

</application>

</manifest>

Following will be the content of **res/layout/activity\_main.xml** file to include two buttons −

<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 services"

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="Start Services"

android:onClick="startService"

android:layout\_below="@+id/imageButton"

android:layout\_centerHorizontal="true" />

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Stop Services"

android:id="@+id/button"

android:onClick="stopService"

android:layout\_below="@+id/button2"

android:layout\_alignLeft="@+id/button2"

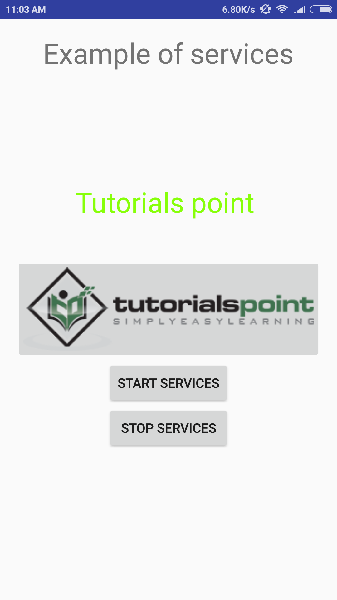
android:layout\_alignStart="@+id/button2"

android:layout\_alignRight="@+id/button2"

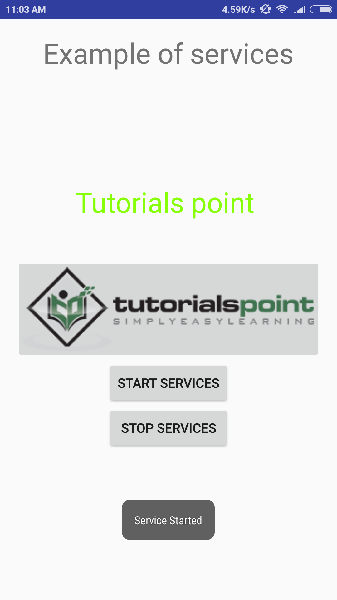
android:layout\_alignEnd="@+id/button2" />

</RelativeLayout>

Let's try to run our modified **Hello World!** application we just modified. I assume you had created your **AVD** while doing environment setup. To run the app from Android studio, open one of your project's activity files and click Run Android StudioRun Iconicon from the tool bar. Android Studio 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 −



Now to start your service, let's click on **Start Service** button, this will start the service and as per our programming in *onStartCommand()* method, a message *Service Started* will appear on the bottom of the the simulator as follows −



To stop the service, you can click the Stop Service button.

# 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 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) {

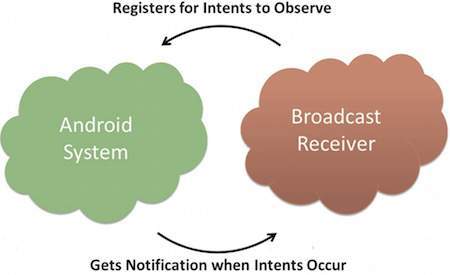
Toast.makeText(context, "Intent Detected.", Toast.LENGTH\_LONG).show();

}

}

## 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.



#### 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.

|  |  |
| --- | --- |
| **Sr.No** | **Event Constant & Description** |
| 1 | **android.intent.action.BATTERY\_CHANGED**  Sticky broadcast containing the charging state, level, and other information about the battery. |
| 2 | **android.intent.action.BATTERY\_LOW**  Indicates low battery condition on the device. |
| 3 | **android.intent.action.BATTERY\_OKAY**  Indicates the battery is now okay after being low. |
| 4 | **android.intent.action.BOOT\_COMPLETED**  This is broadcast once, after the system has finished booting. |
| 5 | **android.intent.action.BUG\_REPORT**  Show activity for reporting a bug. |
| 6 | **android.intent.action.CALL**  Perform a call to someone specified by the data. |
| 7 | **android.intent.action.CALL\_BUTTON**  The user pressed the "call" button to go to the dialer or other appropriate UI for placing a call. |
| 8 | **android.intent.action.DATE\_CHANGED**  The date has changed. |
| 9 | **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 regsitered 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* chapter −

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use Android studio to create an Android application and name it as *My Application* under a package *com.example.tutorialspoint7.myapplication* as explained in the *Hello World Example* chapter. |
| 2 | Modify main activity file *MainActivity.java* to add *broadcastIntent()* method. |
| 3 | Create a new java file called *MyReceiver.java* under the package *com.example.tutorialspoint7.myapplication* to define a BroadcastReceiver. |
| 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 *AndroidManifest.xml* file using <receiver.../> tag |
| 5 | Modify the default content of *res/layout/activity\_main.xml* 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 **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.tutorialspoint7.myapplication;

import android.app.Activity;

import android.content.Intent;

import android.os.Bundle;

import android.view.View;

public class MainActivity extends Activity {

/\*\* Called when the activity is first created. \*/

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

}

// 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 **MyReceiver.java**:

package com.example.tutorialspoint7.myapplication;

import android.content.BroadcastReceiver;

import android.content.Context;

import android.content.Intent;

import android.widget.Toast;

/\*\*

\* Created by TutorialsPoint7 on 8/23/2016.

\*/

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 <receiver.../> tag to include our service:

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

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.tutorialspoint7.myapplication">

<application

android:allowBackup="true"

android:icon="@mipmap/ic\_launcher"

android:label="@string/app\_name"

android:supportsRtl="true"

android:theme="@style/AppTheme">

<activity android:name=".MainActivity">

<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>

Let's try to run our modified **Hello World!** application we just modified. I assume you had created your **AVD** while doing environment set-up. To run the app from Android studio, open one of your project's activity files and click Run Eclipse Run Iconicon from the tool bar. Android Studio 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 −



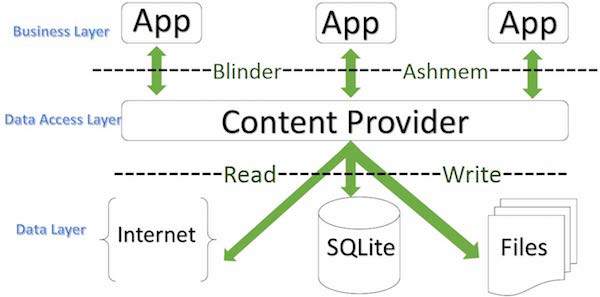
Now to broadcast our custom intent, let's click on **Broadcast Intent** button, this will broadcast our custom intent *"com.tutorialspoint.CUSTOM\_INTENT"* which will be intercepted by our registered BroadcastReceiver i.e. MyReceiver and as per our implemented logic a toast will appear on the bottom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



#### ContentProvider

**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.

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 −

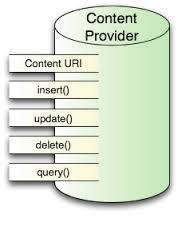
|  |  |
| --- | --- |
| **Sr.No** | **Part & Description** |
| 1 | **prefix**  This is always set to content:// |
| 2 | **authority**  This specifies the name of the content provider, for example *contacts*, *browser* etc. For third-party content providers, this could be the fully qualified name, such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

## 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 *ContentProviderbaseclass.*
* 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 −



#### ContentProvider

* **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.

## Example

This example will explain you how to create your own *ContentProvider*. So let's follow the following steps to similar to what we followed while creating *Hello World Example*−

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use Android StudioIDE to create an Android application and name it as *My Application* under a package *com.example.MyApplication*, with blank Activity. |
| 2 | Modify main activity file *MainActivity.java* to add two new methods *onClickAddName()* and *onClickRetrieveStudents()*. |
| 3 | Create a new java file called *StudentsProvider.java* under the package *com.example.MyApplication* to define your actual provider and associated methods. |
| 4 | Register your content provider in your *AndroidManifest.xml* file using <provider.../> tag |
| 5 | Modify the default content of *res/layout/activity\_main.xml* file to include a small GUI to add students records. |
| 6 | No need to change string.xml.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.MyApplication/MainActivity.java**. This file can include each of the fundamental life cycle methods. We have added two new methods *onClickAddName()* and *onClickRetrieveStudents()* to handle user interaction with the application.

package com.example.MyApplication;

import android.net.Uri;

import android.os.Bundle;

import android.app.Activity;

import android.content.ContentValues;

import android.content.CursorLoader;

import android.database.Cursor;

import android.view.Menu;

import android.view.View;

import android.widget.EditText;

import android.widget.Toast;

public class MainActivity extends Activity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

}

public void onClickAddName(View view) {

// Add a new student record

ContentValues values = new ContentValues();

values.put(StudentsProvider.NAME,

((EditText)findViewById(R.id.editText2)).getText().toString());

values.put(StudentsProvider.GRADE,

((EditText)findViewById(R.id.editText3)).getText().toString());

Uri uri = getContentResolver().insert(

StudentsProvider.CONTENT\_URI, values);

Toast.makeText(getBaseContext(),

uri.toString(), Toast.LENGTH\_LONG).show();

}

public void onClickRetrieveStudents(View view) {

// Retrieve student records

String URL = "content://com.example.MyApplication.StudentsProvider";

Uri students = Uri.parse(URL);

Cursor c = managedQuery(students, null, null, null, "name");

if (c.moveToFirst()) {

do{

Toast.makeText(this,

c.getString(c.getColumnIndex(StudentsProvider.\_ID)) +

", " + c.getString(c.getColumnIndex( StudentsProvider.NAME)) +

", " + c.getString(c.getColumnIndex( StudentsProvider.GRADE)),

Toast.LENGTH\_SHORT).show();

} while (c.moveToNext());

}

}

}

Create new file StudentsProvider.java under *com.example.MyApplication* package and following is the content of **src/com.example.MyApplication/StudentsProvider.java** −

package com.example.MyApplication;

import java.util.HashMap;

import android.content.ContentProvider;

import android.content.ContentUris;

import android.content.ContentValues;

import android.content.Context;

import android.content.UriMatcher;

import android.database.Cursor;

import android.database.SQLException;

import android.database.sqlite.SQLiteDatabase;

import android.database.sqlite.SQLiteOpenHelper;

import android.database.sqlite.SQLiteQueryBuilder;

import android.net.Uri;

import android.text.TextUtils;

public class StudentsProvider extends ContentProvider {

static final String PROVIDER\_NAME = "com.example.MyApplication.StudentsProvider";

static final String URL = "content://" + PROVIDER\_NAME + "/students";

static final Uri CONTENT\_URI = Uri.parse(URL);

static final String \_ID = "\_id";

static final String NAME = "name";

static final String GRADE = "grade";

private static HashMap<String, String> STUDENTS\_PROJECTION\_MAP;

static final int STUDENTS = 1;

static final int STUDENT\_ID = 2;

static final UriMatcher uriMatcher;

static{

uriMatcher = new UriMatcher(UriMatcher.NO\_MATCH);

uriMatcher.addURI(PROVIDER\_NAME, "students", STUDENTS);

uriMatcher.addURI(PROVIDER\_NAME, "students/#", STUDENT\_ID);

}

/\*\*

\* Database specific constant declarations

\*/

private SQLiteDatabase db;

static final String DATABASE\_NAME = "College";

static final String STUDENTS\_TABLE\_NAME = "students";

static final int DATABASE\_VERSION = 1;

static final String CREATE\_DB\_TABLE =

" CREATE TABLE " + STUDENTS\_TABLE\_NAME +

" (\_id INTEGER PRIMARY KEY AUTOINCREMENT, " +

" name TEXT NOT NULL, " +

" grade TEXT NOT NULL);";

/\*\*

\* Helper class that actually creates and manages

\* the provider's underlying data repository.

\*/

private static class DatabaseHelper extends SQLiteOpenHelper {

DatabaseHelper(Context context){

super(context, DATABASE\_NAME, null, DATABASE\_VERSION);

}

@Override

public void onCreate(SQLiteDatabase db) {

db.execSQL(CREATE\_DB\_TABLE);

}

@Override

public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {

db.execSQL("DROP TABLE IF EXISTS " + STUDENTS\_TABLE\_NAME);

onCreate(db);

}

}

@Override

public boolean onCreate() {

Context context = getContext();

DatabaseHelper dbHelper = new DatabaseHelper(context);

/\*\*

\* Create a write able database which will trigger its

\* creation if it doesn't already exist.

\*/

db = dbHelper.getWritableDatabase();

return (db == null)? false:true;

}

@Override

public Uri insert(Uri uri, ContentValues values) {

/\*\*

\* Add a new student record

\*/

long rowID = db.insert( STUDENTS\_TABLE\_NAME, "", values);

/\*\*

\* If record is added successfully

\*/

if (rowID > 0) {

Uri \_uri = ContentUris.withAppendedId(CONTENT\_URI, rowID);

getContext().getContentResolver().notifyChange(\_uri, null);

return \_uri;

}

throw new SQLException("Failed to add a record into " + uri);

}

@Override

public Cursor query(Uri uri, String[] projection,

String selection,String[] selectionArgs, String sortOrder) {

SQLiteQueryBuilder qb = new SQLiteQueryBuilder();

qb.setTables(STUDENTS\_TABLE\_NAME);

switch (uriMatcher.match(uri)) {

case STUDENTS:

qb.setProjectionMap(STUDENTS\_PROJECTION\_MAP);

break;

case STUDENT\_ID:

qb.appendWhere( \_ID + "=" + uri.getPathSegments().get(1));

break;

default:

}

if (sortOrder == null || sortOrder == ""){

/\*\*

\* By default sort on student names

\*/

sortOrder = NAME;

}

Cursor c = qb.query(db, projection, selection,

selectionArgs,null, null, sortOrder);

/\*\*

\* register to watch a content URI for changes

\*/

c.setNotificationUri(getContext().getContentResolver(), uri);

return c;

}

@Override

public int delete(Uri uri, String selection, String[] selectionArgs) {

int count = 0;

switch (uriMatcher.match(uri)){

case STUDENTS:

count = db.delete(STUDENTS\_TABLE\_NAME, selection, selectionArgs);

break;

case STUDENT\_ID:

String id = uri.getPathSegments().get(1);

count = db.delete( STUDENTS\_TABLE\_NAME, \_ID + " = " + id +

(!TextUtils.isEmpty(selection) ? "

AND (" + selection + ')' : ""), selectionArgs);

break;

default:

throw new IllegalArgumentException("Unknown URI " + uri);

}

getContext().getContentResolver().notifyChange(uri, null);

return count;

}

@Override

public int update(Uri uri, ContentValues values,

String selection, String[] selectionArgs) {

int count = 0;

switch (uriMatcher.match(uri)) {

case STUDENTS:

count = db.update(STUDENTS\_TABLE\_NAME, values, selection, selectionArgs);

break;

case STUDENT\_ID:

count = db.update(STUDENTS\_TABLE\_NAME, values,

\_ID + " = " + uri.getPathSegments().get(1) +

(!TextUtils.isEmpty(selection) ? "

AND (" +selection + ')' : ""), selectionArgs);

break;

default:

throw new IllegalArgumentException("Unknown URI " + uri );

}

getContext().getContentResolver().notifyChange(uri, null);

return count;

}

@Override

public String getType(Uri uri) {

switch (uriMatcher.match(uri)){

/\*\*

\* Get all student records

\*/

case STUDENTS:

return "vnd.android.cursor.dir/vnd.example.students";

/\*\*

\* Get a particular student

\*/

case STUDENT\_ID:

return "vnd.android.cursor.item/vnd.example.students";

default:

throw new IllegalArgumentException("Unsupported URI: " + uri);

}

}

}

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

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

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.MyApplication">

<application

android:allowBackup="true"

android:icon="@mipmap/ic\_launcher"

android:label="@string/app\_name"

android:supportsRtl="true"

android:theme="@style/AppTheme">

<activity android:name=".MainActivity">

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

<provider android:name="StudentsProvider"

android:authorities="com.example.MyApplication.StudentsProvider"/>

</application>

</manifest>

Following will be the content of **res/layout/activity\_main.xml** file−

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

<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:paddingBottom="@dimen/activity\_vertical\_margin"

android:paddingLeft="@dimen/activity\_horizontal\_margin"

android:paddingRight="@dimen/activity\_horizontal\_margin"

android:paddingTop="@dimen/activity\_vertical\_margin"

tools:context="com.example.MyApplication.MainActivity">

<TextView

android:id="@+id/textView1"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Content provider"

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\_below="@+id/textView1"

android:layout\_centerHorizontal="true" />

<ImageButton

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/imageButton"

android:src="@drawable/abc"

android:layout\_below="@+id/textView2"

android:layout\_centerHorizontal="true" />

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/button2"

android:text="Add Name"

android:layout\_below="@+id/editText3"

android:layout\_alignRight="@+id/textView2"

android:layout\_alignEnd="@+id/textView2"

android:layout\_alignLeft="@+id/textView2"

android:layout\_alignStart="@+id/textView2"

android:onClick="onClickAddName"/>

<EditText

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/editText"

android:layout\_below="@+id/imageButton"

android:layout\_alignRight="@+id/imageButton"

android:layout\_alignEnd="@+id/imageButton" />

<EditText

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/editText2"

android:layout\_alignTop="@+id/editText"

android:layout\_alignLeft="@+id/textView1"

android:layout\_alignStart="@+id/textView1"

android:layout\_alignRight="@+id/textView1"

android:layout\_alignEnd="@+id/textView1"

android:hint="Name"

android:textColorHint="@android:color/holo\_blue\_light" />

<EditText

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/editText3"

android:layout\_below="@+id/editText"

android:layout\_alignLeft="@+id/editText2"

android:layout\_alignStart="@+id/editText2"

android:layout\_alignRight="@+id/editText2"

android:layout\_alignEnd="@+id/editText2"

android:hint="Grade"

android:textColorHint="@android:color/holo\_blue\_bright" />

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Retrive student"

android:id="@+id/button"

android:layout\_below="@+id/button2"

android:layout\_alignRight="@+id/editText3"

android:layout\_alignEnd="@+id/editText3"

android:layout\_alignLeft="@+id/button2"

android:layout\_alignStart="@+id/button2"

android:onClick="onClickRetrieveStudents"/>

</RelativeLayout>

Make sure you have following content of **res/values/strings.xml** file:

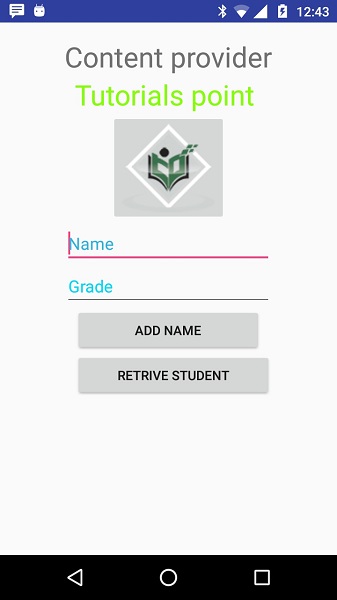
<?xml version="1.0" encoding="utf-8"?>

<resources>

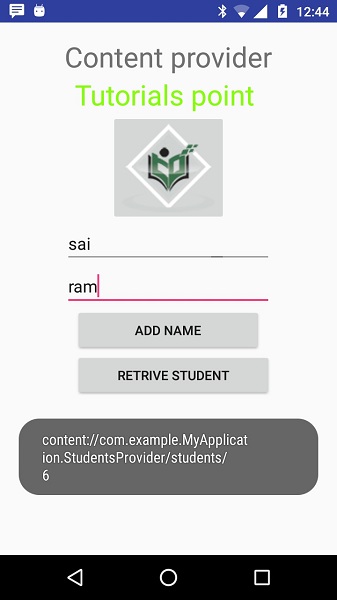
<string name="app\_name">My Application</string>

</resources>;

Let's try to run our modified **My Application** application we just created. I assume you had created your **AVD** while doing environment set-up. To run the app from Android Studio IDE, open one of your project's activity files and click Run Android StudioRun Iconicon from the tool bar. Android Studio 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, be patience because it may take sometime based on your computer speed −



Now let's enter student **Name** and **Grade** and finally click on **Add Name** button, this will add student record in the database and will flash a message at the bottom showing ContentProvider URI along with record number added in the database. This operation makes use of our **insert()** method. Let's repeat this process to add few more students in the database of our content provider.



Once you are done with adding records in the database, now its time to ask ContentProvider to give us those records back, so let's click **Retrieve Students** button which will fetch and display all the records one by one which is as per our the implementation of our **query()** method.

You can write activities against update and delete operations by providing callback functions in **MainActivity.java** file and then modify user interface to have buttons for update and deleted operations in the same way as we have done for add and read operations.

This way you can use existing Content Provider like Address Book or you can use Content Provider concept in developing nice database oriented applications where you can perform all sort of database operations like read, write, update and delete as explained above in the example.

# 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**.

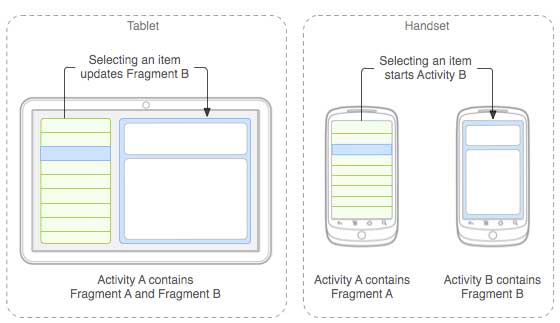
Following are important points about fragment −

* 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.



#### Fragment lifecycle

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](https://www.tutorialspoint.com/android/android_single_fragments.htm) − Single frame fragments are using for hand hold devices like mobiles, here we can show only one fragment as a view.
* [List fragments](https://www.tutorialspoint.com/android/android_list_fragment.htm) − fragments having special list view is called as list fragment
* [Fragments transaction](https://www.tutorialspoint.com/android/android_fragment_transitions.htm) − 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 | **Context.startActivity()**  The Intent object is passed to this method to launch a new activity or get an existing activity to do something new. |
| 2 | **Context.startService()**  The Intent object is passed to this method to initiate a service or deliver new instructions to an ongoing service. |
| 3 | **Context.sendBroadcast()**  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](https://www.tutorialspoint.com/android/android_intent_standard_actions.htm)

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 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().

Some examples of action/data pairs are −

|  |  |
| --- | --- |
| **Sr.No.** | **Action/Data Pair & Description** |
| 1 | **ACTION\_VIEW content://contacts/people/1**  Display information about the person whose identifier is "1". |
| 2 | **ACTION\_DIAL content://contacts/people/1**  Display the phone dialer with the person filled in. |
| 3 | **ACTION\_VIEW tel:123**  Display the phone dialer with the given number filled in. |
| 4 | **ACTION\_DIAL tel:123**  Display the phone dialer with the given number filled in. |
| 5 | **ACTION\_EDIT content://contacts/people/1**  Edit information about the person whose identifier is "1". |
| 6 | **ACTION\_VIEW content://contacts/people/**  Display a list of people, which the user can browse through. |
| 7 | **ACTION\_SET\_WALLPAPER**  Show settings for choosing wallpaper |
| 8 | **ACTION\_SYNC**  It going to be synchronous the data,Constant Value is **android.intent.action.SYNC** |
| 9 | **ACTION\_SYSTEM\_TUTORIAL**  It will start the platform-defined tutorial(Default tutorial or start up tutorial) |
| 10 | **ACTION\_TIMEZONE\_CHANGED**  It intimates when time zone has changed |
| 11 | **ACTION\_UNINSTALL\_PACKAGE**  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](https://www.tutorialspoint.com/android/android_intent_standard_categories.htm).

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](https://www.tutorialspoint.com/android/android_intent_standard_extra_data.htm)

### 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 | **FLAG\_ACTIVITY\_CLEAR\_TASK**  If set in an Intent passed to Context.startActivity(), 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 FLAG\_ACTIVITY\_NEW\_TASK. |
| 2 | **FLAG\_ACTIVITY\_CLEAR\_TOP**  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 | **FLAG\_ACTIVITY\_NEW\_TASK**  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().

## 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");

## Example

Following example shows the functionality of a Android Intent to launch various Android built-in applications.

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use Android studio IDE to create an Android application and name it as *My Application* under a package *com.example.saira\_000.myapplication*. |
| 2 | Modify *src/main/java/MainActivity.java* file and add the code to define two listeners corresponding two buttons ie. Start Browser and Start Phone. |
| 3 | Modify layout XML file *res/layout/activity\_main.xml* to add three buttons in linear layout. |
| 4 | 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**.

package com.example.saira\_000.myapplication;

import android.content.Intent;

import android.net.Uri;

import android.support.v7.app.AppCompatActivity;

import android.os.Bundle;

import android.view.View;

import android.widget.Button;

public class MainActivity extends AppCompatActivity {

Button b1,b2;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

b1=(Button)findViewById(R.id.button);

b1.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

Intent i = new Intent(android.content.Intent.ACTION\_VIEW,

Uri.parse("http://www.example.com"));

startActivity(i);

}

});

b2=(Button)findViewById(R.id.button2);

b2.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

Intent i = new Intent(android.content.Intent.ACTION\_VIEW,

Uri.parse("tel:9510300000"));

startActivity(i);

}

});

}

}

Following will be the content of **res/layout/activity\_main.xml** file −

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

<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="Intent Example"

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\_below="@+id/textView1"

android:layout\_centerHorizontal="true" />

<ImageButton

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/imageButton"

android:src="@drawable/abc"

android:layout\_below="@+id/textView2"

android:layout\_centerHorizontal="true" />

<EditText

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/editText"

android:layout\_below="@+id/imageButton"

android:layout\_alignRight="@+id/imageButton"

android:layout\_alignEnd="@+id/imageButton" />

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Start Browser"

android:id="@+id/button"

android:layout\_alignTop="@+id/editText"

android:layout\_alignRight="@+id/textView1"

android:layout\_alignEnd="@+id/textView1"

android:layout\_alignLeft="@+id/imageButton"

android:layout\_alignStart="@+id/imageButton" />

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Start Phone"

android:id="@+id/button2"

android:layout\_below="@+id/button"

android:layout\_alignLeft="@+id/button"

android:layout\_alignStart="@+id/button"

android:layout\_alignRight="@+id/textView2"

android:layout\_alignEnd="@+id/textView2" />

</RelativeLayout>

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

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

<resources>

<string name="app\_name">My Applicaiton</string>

</resources>

Following is the default content of **AndroidManifest.xml** −

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

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.saira\_000.myapplication">

<application

android:allowBackup="true"

android:icon="@mipmap/ic\_launcher"

android:label="@string/app\_name"

android:supportsRtl="true"

android:theme="@style/AppTheme">

<activity android:name=".MainActivity">

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

</application>

</manifest>

Let's try to run your **My Application** application. I assume you had created your **AVD** while doing environment setup. To run the app from Android Studio, open one of your project's activity files and click Run icon from the toolbar.Android Studio installs the app on your AVD and starts it and if everything is fine with your setup and application, it will display following Emulator window −

Now click on **Start Browser** button, which will start a browser configured and display http://www.example.com as shown below −

Similar way you can launch phone interface using Start Phone button, which will allow you to dial already given phone number.

## 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.

## Example

Following example is a modification of the above example. Here we will see how Android resolves conflict if one intent is invoking two activities defined in , next how to invoke a custom activity using a filter and third one is an exception case if Android does not file appropriate activity defined for an intent.

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use android studio to create an Android application and name it as *My Application* under a package *com.example.tutorialspoint7.myapplication;*. |
| 2 | Modify *src/Main/Java/MainActivity.java* file and add the code to define three listeners corresponding to three buttons defined in layout file. |
| 3 | Add a new *src/Main/Java/CustomActivity.java* file to have one custom activity which will be invoked by different intents. |
| 4 | Modify layout XML file *res/layout/activity\_main.xml* to add three buttons in linear layout. |
| 5 | Add one layout XML file *res/layout/custom\_view.xml* to add a simple <TextView> to show the passed data through intent. |
| 6 | Modify *AndroidManifest.xml* to add <intent-filter> to define rules for your intent to invoke custom activity. |
| 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/MainActivity.java**.

package com.example.tutorialspoint7.myapplication;

import android.content.Intent;

import android.net.Uri;

import android.support.v7.app.AppCompatActivity;

import android.os.Bundle;

import android.view.View;

import android.widget.Button;

public class MainActivity extends AppCompatActivity {

Button b1,b2,b3;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

b1=(Button)findViewById(R.id.button);

b1.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

Intent i = new Intent(android.content.Intent.ACTION\_VIEW,

Uri.parse("http://www.example.com"));

startActivity(i);

}

});

b2 = (Button)findViewById(R.id.button2);

b2.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

Intent i = new Intent("com.example.

tutorialspoint7.myapplication.

LAUNCH",Uri.parse("http://www.example.com"));

startActivity(i);

}

});

b3 = (Button)findViewById(R.id.button3);

b3.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

Intent i = new Intent("com.example.

My Application.LAUNCH",

Uri.parse("https://www.example.com"));

startActivity(i);

}

});

}

}

Following is the content of the modified main activity file **src/com.example.My Application/CustomActivity.java**.

package com.example.tutorialspoint7.myapplication;

import android.app.Activity;

import android.net.Uri;

import android.os.Bundle;

import android.widget.TextView;

/\*\*

\* Created by TutorialsPoint7 on 8/23/2016.

\*/

public class CustomActivity extends Activity {

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.custom\_view);

TextView label = (TextView) findViewById(R.id.show\_data);

Uri url = getIntent().getData();

label.setText(url.toString());

}

}

Following will be the content of **res/layout/activity\_main.xml** file −

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

<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:paddingBottom="@dimen/activity\_vertical\_margin"

android:paddingLeft="@dimen/activity\_horizontal\_margin"

android:paddingRight="@dimen/activity\_horizontal\_margin"

android:paddingTop="@dimen/activity\_vertical\_margin"

tools:context="com.example.tutorialspoint7.myapplication.MainActivity">

<TextView

android:id="@+id/textView1"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Intent Example"

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\_below="@+id/textView1"

android:layout\_centerHorizontal="true" />

<ImageButton

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/imageButton"

android:src="@drawable/abc"

android:layout\_below="@+id/textView2"

android:layout\_centerHorizontal="true" />

<EditText

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/editText"

android:layout\_below="@+id/imageButton"

android:layout\_alignRight="@+id/imageButton"

android:layout\_alignEnd="@+id/imageButton" />

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Start Browser"

android:id="@+id/button"

android:layout\_alignTop="@+id/editText"

android:layout\_alignLeft="@+id/imageButton"

android:layout\_alignStart="@+id/imageButton"

android:layout\_alignEnd="@+id/imageButton" />

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Start browsing with launch action"

android:id="@+id/button2"

android:layout\_below="@+id/button"

android:layout\_alignLeft="@+id/button"

android:layout\_alignStart="@+id/button"

android:layout\_alignEnd="@+id/button" />

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Exceptional condition"

android:id="@+id/button3"

android:layout\_below="@+id/button2"

android:layout\_alignLeft="@+id/button2"

android:layout\_alignStart="@+id/button2"

android:layout\_toStartOf="@+id/editText"

android:layout\_alignParentEnd="true" />

</RelativeLayout>

Following will be the content of **res/layout/custom\_view.xml** file −

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

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:orientation="vertical" android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<TextView android:id="@+id/show\_data"

android:layout\_width="fill\_parent"

android:layout\_height="400dp"/>

</LinearLayout>

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

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

<resources>

<string name="app\_name">My Application</string>

</resources>

Following is the default content of **AndroidManifest.xml** −

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

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.tutorialspoint7.myapplication">

<application

android:allowBackup = "true"

android:icon = "@mipmap/ic\_launcher"

android:label = "@string/app\_name"

android:supportsRtl = "true"

android:theme = "@style/AppTheme">

<activity android:name = ".MainActivity">

<intent-filter>

<action android:name = "android.intent.action.MAIN" />

<category android:name = "android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

<activity android:name="com.example.tutorialspoint7.myapplication.CustomActivity">

<intent-filter>

<action android:name = "android.intent.action.VIEW" />

<action android:name = "com.example.tutorialspoint7.myapplication.LAUNCH" />

<category android:name = "android.intent.category.DEFAULT" />

<data android:scheme = "http" />

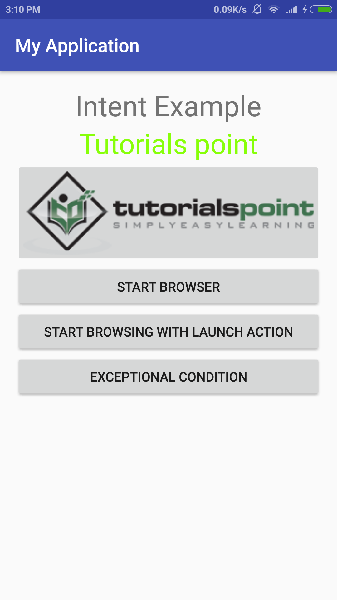
</intent-filter>

</activity>

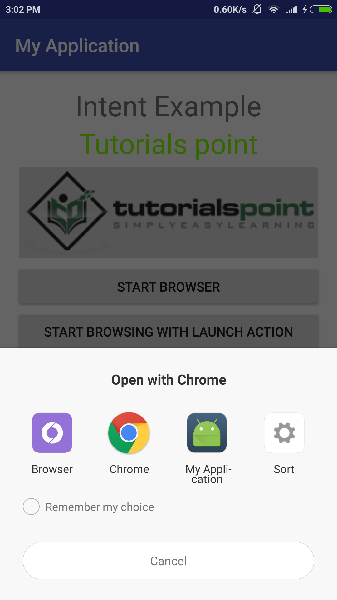
</application>

</manifest>

Let's try to run your **My Application** application. I assume you had created your **AVD** while doing environment setup. To run the app from Android Studio, open one of your project's activity files and click Run Eclipse Run Iconicon from the toolbar. Android Studio installs the app on your AVD and starts it and if everything is fine with your setup and application, it will display following Emulator window −



Now let's start with first button "Start Browser with VIEW Action". Here we have defined our custom activity with a filter "android.intent.action.VIEW", and there is already one default activity against VIEW action defined by Android which is launching web browser, So android displays following two options to select the activity you want to launch.

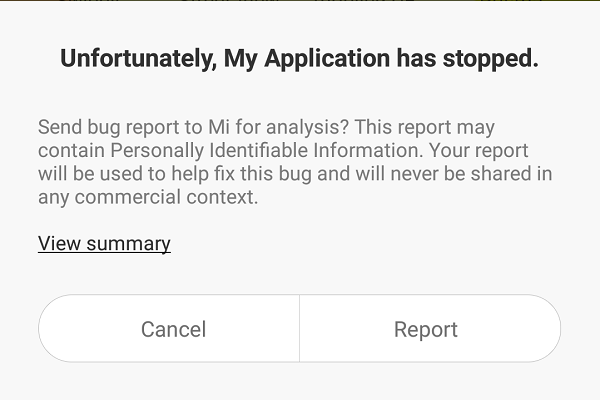


Now if you select Browser, then Android will launch web browser and open example.com website but if you select IndentDemo option then Android will launch CustomActivity which does nothing but just capture passed data and displays in a text view as follows −



Now go back using back button and click on "Start Browser with LAUNCH Action" button, here Android applies filter to choose define activity and it simply launch your custom activity

Again, go back using back button and click on "Exception Condition" button, here Android tries to find out a valid filter for the given intent but it does not find a valid activity defined because this time we have used data as **https** instead of **http** though we are giving a correct action, so Android raises an exception and shows following screen −



# 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.

#### Layout params

This tutorial is more about creating your GUI based on layouts defined in XML file. 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 | [Linear Layout](https://www.tutorialspoint.com/android/android_linear_layout.htm)  LinearLayout is a view group that aligns all children in a single direction, vertically or horizontally. |
| 2 | [Relative Layout](https://www.tutorialspoint.com/android/android_relative_layout.htm)  RelativeLayout is a view group that displays child views in relative positions. |
| 3 | [Table Layout](https://www.tutorialspoint.com/android/android_table_layout.htm)  TableLayout is a view that groups views into rows and columns. |
| 4 | [Absolute Layout](https://www.tutorialspoint.com/android/android_absolute_layout.htm)  AbsoluteLayout enables you to specify the exact location of its children. |
| 5 | [Frame Layout](https://www.tutorialspoint.com/android/android_frame_layout.htm)  The FrameLayout is a placeholder on screen that you can use to display a single view. |
| 6 | [List View](https://www.tutorialspoint.com/android/android_list_view.htm)  ListView is a view group that displays a list of scrollable items. |
| 7 | [Grid View](https://www.tutorialspoint.com/android/android_grid_view.htm)  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 | **android:id**  This is the ID which uniquely identifies the view. |
| 2 | **android:layout\_width**  This is the width of the layout. |
| 3 | **android:layout\_height**  This is the height of the layout |
| 4 | **android:layout\_marginTop**  This is the extra space on the top side of the layout. |
| 5 | **android:layout\_marginBottom**  This is the extra space on the bottom side of the layout. |
| 6 | **android:layout\_marginLeft**  This is the extra space on the left side of the layout. |
| 7 | **android:layout\_marginRight**  This is the extra space on the right side of the layout. |
| 8 | **android:layout\_gravity**  This specifies how child Views are positioned. |
| 9 | **android:layout\_weight**  This specifies how much of the extra space in the layout should be allocated to the View. |
| 10 | **android:layout\_x**  This specifies the x-coordinate of the layout. |
| 11 | **android:layout\_y**  This specifies the y-coordinate of the layout. |
| 12 | **android:layout\_width**  This is the width of the layout. |
| 13 | **android:layout\_width**  This is the width of the layout. |
| 14 | **android:paddingLeft**  This is the left padding filled for the layout. |
| 15 | **android:paddingRight**  This is the right padding filled for the layout. |
| 16 | **android:paddingTop**  This is the top padding filled for the layout. |
| 17 | **android:paddingBottom**  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 | 0x00800003 | Push object to the beginning of its container, not changing its size. |
| end | 0x00800005 | 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"

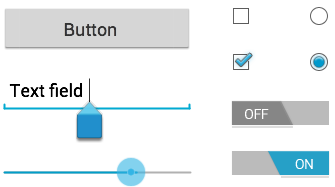
Following is a brief description of @ and + signs −

* 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.



#### UI Elements

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 | [TextView](https://www.tutorialspoint.com/android/android_textview_control.htm)  This control is used to display text to the user. |
| 2 | [EditText](https://www.tutorialspoint.com/android/android_edittext_control.htm)  EditText is a predefined subclass of TextView that includes rich editing capabilities. |
| 3 | [AutoCompleteTextView](https://www.tutorialspoint.com/android/android_autocompletetextview_control.htm)  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 | [Button](https://www.tutorialspoint.com/android/android_button_control.htm)  A push-button that can be pressed, or clicked, by the user to perform an action. |
| 5 | [ImageButton](https://www.tutorialspoint.com/android/android_imagebutton_control.htm)  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 | [CheckBox](https://www.tutorialspoint.com/android/android_checkbox_control.htm)  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 | [ToggleButton](https://www.tutorialspoint.com/android/android_togglebutton_control.htm)  An on/off button with a light indicator. |
| 8 | [RadioButton](https://www.tutorialspoint.com/android/android_radiobutton_control.htm)  The RadioButton has two states: either checked or unchecked. |
| 9 | [RadioGroup](https://www.tutorialspoint.com/android/android_radiogroup_control.htm)  A RadioGroup is used to group together one or more RadioButtons. |
| 10 | [ProgressBar](https://www.tutorialspoint.com/android/android_progressbar.htm)  The ProgressBar view provides visual feedback about some ongoing tasks, such as when you are performing a task in the background. |
| 11 | [Spinner](https://www.tutorialspoint.com/android/android_spinner_control.htm)  A drop-down list that allows users to select one value from a set. |
| 12 | [TimePicker](https://www.tutorialspoint.com/android/android_timepicker_control.htm)  The TimePicker view enables users to select a time of the day, in either 24-hour mode or AM/PM mode. |
| 13 | [DatePicker](https://www.tutorialspoint.com/android/android_datepicker_control.htm)  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);

# Android - 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.

## Event Listeners & Event Handlers

|  |  |
| --- | --- |
| **Event Handler** | **Event Listener & Description** |
| onClick() | **OnClickListener()**  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() | **OnLongClickListener()**  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() | **OnFocusChangeListener()**  This is called when the widget looses its focus ie. user goes away from the view item. You will use onFocusChange() event handler to handle such event. |
| onKey() | **OnFocusChangeListener()**  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() | **OnTouchListener()**  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() | **OnMenuItemClickListener()**  This is called when the user selects a menu item. You will use onMenuItemClick() event handler to handle such event. |
| onCreateContextMenu() | **onCreateContextMenuItemListener()**  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:foucsUp="@=id/button\_l"

## onTouchEvent()

public boolean onTouchEvent(motionEvent event){

switch(event.getAction()){

case TOUCH\_DOWN:

Toast.makeText(this,"you have clicked down Touch button",Toast.LENTH\_LONG).show();

break();

case TOUCH\_UP:

Toast.makeText(this,"you have clicked up touch button",Toast.LENTH\_LONG).show();

break;

case TOUCH\_MOVE:

Toast.makeText(this,"you have clicked move touch button"Toast.LENTH\_LONG).show();

break;

}

return super.onTouchEvent(event) ;

}

## Event Handling Examples

### Event Listeners Registration Using an Anonymous Inner Class

Here you will create an anonymous implementation of the listener and will be useful if each class is applied to a single control only and you have advantage to pass arguments to event handler. In this approach event handler methods can access private data of Activity. No reference is needed to call to Activity.

But if you applied the handler to more than one control, you would have to cut and paste the code for the handler and if the code for the handler is long, it makes the code harder to maintain.

Following are the simple steps to show how we will make use of separate Listener class to register and capture click event. Similar way you can implement your listener for any other required event type.

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use Android studio IDE to create an Android application and name it as *myapplication* under a package *com.example.myapplication* as explained in the *Hello World Example* chapter. |
| 2 | Modify *src/MainActivity.java* file to add click event listeners and handlers for the two buttons defined. |
| 3 | Modify the detault content of *res/layout/activity\_main.xml* file to include Android UI controls. |
| 4 | No need to declare default string constants.Android studio takes care default constants. |
| 5 | Run the application to launch Android emulator and verify the result of the changes done in the aplication. |

Following is the content of the modified main activity file **src/com.example.myapplication/MainActivity.java**. This file can include each of the fundamental lifecycle methods.

package com.example.myapplication;

import android.app.ProgressDialog;

import android.os.Bundle;

import android.support.v7.app.ActionBarActivity;

import android.view.View;

import android.widget.Button;

import android.widget.TextView;

public class MainActivity extends ActionBarActivity {

private ProgressDialog progress;

Button b1,b2;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

progress = new ProgressDialog(this);

b1=(Button)findViewById(R.id.button);

b2=(Button)findViewById(R.id.button2);

b1.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

TextView txtView = (TextView) findViewById(R.id.textView);

txtView.setTextSize(25);

}

});

b2.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

TextView txtView = (TextView) findViewById(R.id.textView);

txtView.setTextSize(55);

}

});

}

}

Following will be the content of **res/layout/activity\_main.xml** file −

Here abc indicates about tutorialspoint logo

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

<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:paddingBottom="@dimen/activity\_vertical\_margin"

android:paddingLeft="@dimen/activity\_horizontal\_margin"

android:paddingRight="@dimen/activity\_horizontal\_margin"

android:paddingTop="@dimen/activity\_vertical\_margin"

tools:context=".MainActivity">

<TextView

android:id="@+id/textView1"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Event Handling "

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:text="Small font"

android:id="@+id/button"

android:layout\_below="@+id/imageButton"

android:layout\_centerHorizontal="true" />

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Large Font"

android:id="@+id/button2"

android:layout\_below="@+id/button"

android:layout\_alignRight="@+id/button"

android:layout\_alignEnd="@+id/button" />

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Hello World!"

android:id="@+id/textView"

android:layout\_below="@+id/button2"

android:layout\_centerHorizontal="true"

android:textSize="25dp" />

</RelativeLayout>

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

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

<resources>

<string name="app\_name">myapplication</string>

</resources>

Following is the default content of **AndroidManifest.xml** −

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

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.myapplication" >

<application

android:allowBackup="true"

android:icon="@drawable/ic\_launcher"

android:label="@string/app\_name"

android:theme="@style/AppTheme" >

<activity

android:name="com.example.myapplication.MainActivity"

android:label="@string/app\_name" >

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

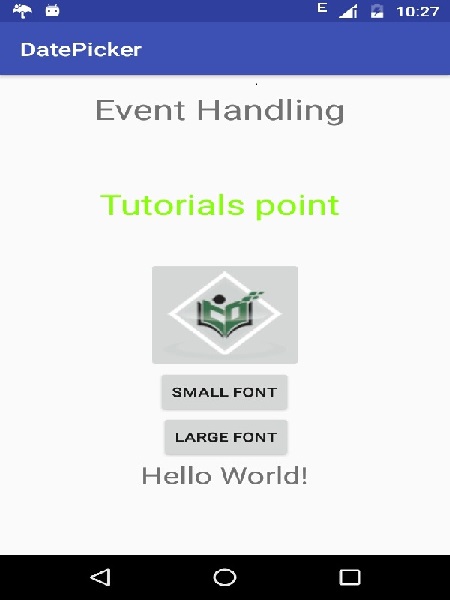
</intent-filter>

</activity>

</application>

</manifest>

Let's try to run your **myapplication** application. I assume you had created your **AVD** while doing environment setup. To run the app from Android Studio, open one of your project's activity files and click Run Eclipse Run Iconicon from the toolbar. Android Studio installs the app on your AVD and starts it and if everything is fine with your setup and application, it will display following Emulator window −



Now you try to click on two buttons, one by one and you will see that font of the **Hello World** text will change, which happens because registered click event handler method is being called against each click event.

## Exercise

I will recommend to try writing different event handlers for different event types and understand exact difference in different event types and their handling. Events related to menu, spinner, pickers widgets are little different but they are also based on the same concepts as explained above.

# Android - 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>

To understand the concept related to Android Style, you can check [Style Demo Example](https://www.tutorialspoint.com/android/android_style_demo_example.htm).

## 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

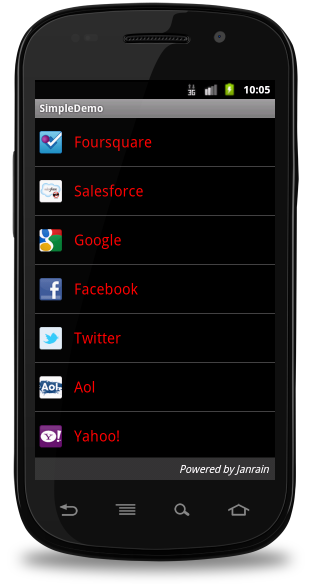
android:name="com.myapp.MyActivity"

...

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" ...>

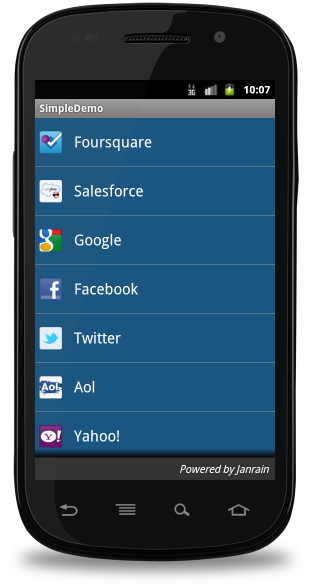
<item name="android:windowBackground">@color/my\_custom\_color</item>

<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.

Nine Patch Button

#### 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=...>

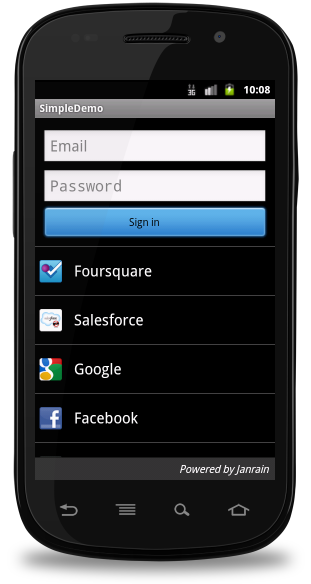
...

<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">

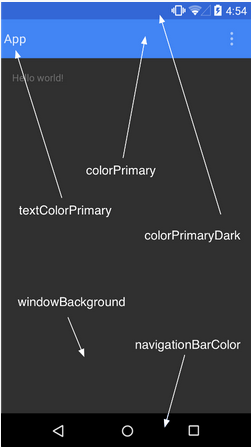
...

</style>

To understand the concept related to Android Theme, you can check [Theme Demo Example](https://www.tutorialspoint.com/android/android_theme_demo_example.htm).

## Styling the colour 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)](https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/core/res/res/values/styles.xml)
* [Android Themes (themes.xml)](https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/core/res/res/values/themes.xml)

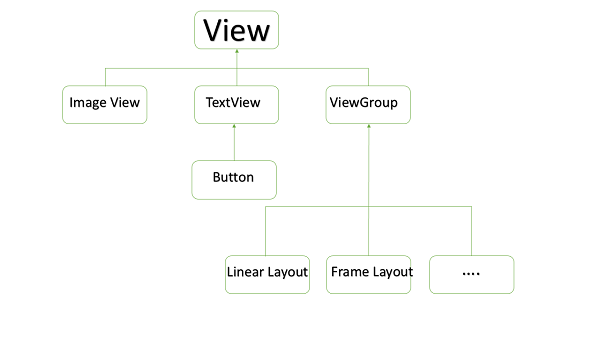
# Android - Custom Components

Implementing own components in pre built-in components with extending subclass with own defined class

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.

This tutorial explains you how to create custom Views and use them in your application using simple and easy steps.



#### Example of Custom Components in Custom View hierarchy

## Creating a Simple Custom Component

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use Android studio IDE to create an Android application and name it as *myapplication* under a package *com.example.tutorialspoint7.myapplication* as explained in the *Hello World Example* chapter. |
| 2 | Create an XML *res/values/attrs.xml* file to define new attributes along with their data type. |
| 3 | Create *src/mainactivity.java* file and add the code to define your custom component |
| 4 | Modify *res/layout/activity\_main.xml* file and add the code to create Colour compound view instance along with few default attributes and new attributes. |
| 5 | Run the application to launch Android emulator and verify the result of the changes done in the application. |

Create the following attributes file called attrs.xml in your res/values folder.

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

<resources>

<declare-styleable name="TimeView">

<declare-styleable name="TimeView">

<attr name="title" format="string" />

<attr name="setColor" format="boolean"/>

</declare-styleable>

</declare-styleable>

</resources>

Change the layout file used by the activity to the following.

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

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

xmlns:tools="http://schemas.android.com/tools"

xmlns:custom="http://schemas.android.com/apk/res-auto"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

tools:context=".MainActivity" >

<com.example.tutorialspoint7.myapplication.TimeView

android:id="@+id/timeView"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:textColor="#fff"

android:textSize="40sp"

custom:title="my time view"

custom:setColor="true" />

<TextView

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:id="@+id/simple"

android:layout\_below="@id/timeView"

android:layout\_marginTop="10dp" />

</RelativeLayout>

Create the following java file called timeview for your compound view.

package com.example.tutorialspoint7.myapplication;

/\*\*

\* Created by TutorialsPoint7 on 9/14/2016.

\*/

import java.text.SimpleDateFormat;

import java.util.Calendar;

import android.content.Context;

import android.content.res.TypedArray;

import android.graphics.Color;

import android.util.AttributeSet;

import android.widget.TextView;

public class TimeView extends TextView {

private String titleText;

private boolean color;

public TimeView(Context context) {

super(context);

setTimeView();

}

public TimeView(Context context, AttributeSet attrs) {

super(context, attrs);

// retrieved values correspond to the positions of the attributes

TypedArray typedArray = context.obtainStyledAttributes(attrs,

R.styleable.TimeView);

int count = typedArray.getIndexCount();

try{

for (int i = 0; i < count; ++i) {

int attr = typedArray.getIndex(i);

// the attr corresponds to the title attribute

if(attr == R.styleable.TimeView\_title) {

// set the text from the layout

titleText = typedArray.getString(attr);

setTimeView();

} else if(attr == R.styleable.TimeView\_setColor) {

// set the color of the attr "setColor"

color = typedArray.getBoolean(attr, false);

decorateText();

}

}

}

// the recycle() will be executed obligatorily

finally {

// for reuse

typedArray.recycle();

}

}

public TimeView(Context context, AttributeSet attrs, int defStyle) {

super(context, attrs, defStyle);

setTimeView();

}

private void setTimeView() {

// has the format hour.minuits am/pm

SimpleDateFormat dateFormat = new SimpleDateFormat("hh.mm aa");

String time = dateFormat.format(Calendar.getInstance().getTime());

if(this.titleText != null )

setText(this.titleText+" "+time);

else

setText(time);

}

private void decorateText() {

// when we set setColor attribute to true in the XML layout

if(this.color == true){

// set the characteristics and the color of the shadow

setShadowLayer(4, 2, 2, Color.rgb(250, 00, 250));

setBackgroundColor(Color.CYAN);

} else {

setBackgroundColor(Color.RED);

}

}

}

Change your Main activity java file to the following code and run your application.

package com.example.tutorialspoint7.myapplication;

import android.os.Bundle;

import android.widget.TextView;

import android.app.Activity;

public class MainActivity extends Activity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

TextView simpleText = (TextView) findViewById(R.id.simple);

simpleText.setText("That is a simple TextView");

}

}

The running application should look like the following screen shot.



# Android - Drag and Drop

Android drag/drop framework allows your users to move data from one View to another View in the current layout using a graphical drag and drop gesture. As of **API 11** drag and drop of view onto other views or view groups is supported.The framework includes following three important components to support drag & drop functionality −

* **Drag event class**.
* **Drag listeners**.
* **Helper methods and classes**.

## The Drag/Drop Process

There are basically four steps or states in the drag and drop process −

* **Started** − This event occurs when you start dragging an item in a layout, your application calls *startDrag()* method to tell the system to start a drag. The arguments inside startDrag() method provide the data to be dragged, metadata for this data, and a callback for drawing the drag shadow.

The system first responds by calling back to your application to get a drag shadow. It then displays the drag shadow on the device.

Next, the system sends a drag event with action type *ACTION\_DRAG\_STARTED* to the registered drag event listeners for all the View objects in the current layout.

To continue to receive drag events, including a possible drop event, a drag event listener must return **true**, If the drag event listener returns false, then it will not receive drag events for the current operation until the system sends a drag event with action type ACTION\_DRAG\_ENDED.

* **Continuing** − The user continues the drag. System sends ACTION\_DRAG\_ENTERED action followed by ACTION\_DRAG\_LOCATION action to the registered drag event listener for the View where dragging point enters. The listener may choose to alter its View object's appearance in response to the event or can react by highlighting its View.

The drag event listener receives a ACTION\_DRAG\_EXITED action after the user has moved the drag shadow outside the bounding box of the View.

* **Dropped** − The user releases the dragged item within the bounding box of a View. The system sends the View object's listener a drag event with action type ACTION\_DROP.
* **Ended** − Just after the action type ACTION\_DROP, the system sends out a drag event with action type ACTION\_DRAG\_ENDED to indicate that the drag operation is over.

## The DragEvent Class

The **DragEvent** represents an event that is sent out by the system at various times during a drag and drop operation. This class provides few Constants and important methods which we use during Drag/Drop process.

### Constants

Following are all constants integers available as a part of DragEvent class.

|  |  |
| --- | --- |
| **Sr.No.** | **Constants & Description** |
| 1 | **ACTION\_DRAG\_STARTED**  Signals the start of a drag and drop operation. |
| 2 | **ACTION\_DRAG\_ENTERED**  Signals to a View that the drag point has entered the bounding box of the View. |
| 3 | **ACTION\_DRAG\_LOCATION**  Sent to a View after ACTION\_DRAG\_ENTERED if the drag shadow is still within the View object's bounding box. |
| 4 | **ACTION\_DRAG\_EXITED**  Signals that the user has moved the drag shadow outside the bounding box of the View. |
| 5 | **ACTION\_DROP**  Signals to a View that the user has released the drag shadow, and the drag point is within the bounding box of the View. |
| 6 | **ACTION\_DRAG\_ENDED**  Signals to a View that the drag and drop operation has concluded. |

### Methods

Following are few important and most frequently used methods available as a part of DragEvent class.

|  |  |
| --- | --- |
| **Sr.No.** | **Constants & Description** |
| 1 | **int getAction()**  Inspect the action value of this event.. |
| 2 | **ClipData getClipData()**  Returns the ClipData object sent to the system as part of the call to startDrag(). |
| 3 | **ClipDescription getClipDescription()**  Returns the ClipDescription object contained in the ClipData. |
| 4 | **boolean getResult()**  Returns an indication of the result of the drag and drop operation. |
| 5 | **float getX()**  Gets the X coordinate of the drag point. |
| 6 | **float getY()**  Gets the Y coordinate of the drag point. |
| 7 | **String toString()**  Returns a string representation of this DragEvent object. |

## Listening for Drag Event

If you want any of your views within a Layout should respond Drag event then your view either implements **View.OnDragListener** or setup **onDragEvent(DragEvent)** callback method. When the system calls the method or listener, it passes to them a DragEvent object explained above. You can have both a listener and a callback method for View object. If this occurs, the system first calls the listener and then defined callback as long as listener returns true.

The combination of the *onDragEvent(DragEvent)* method and *View.OnDragListener* is analogous to the combination of the **onTouchEvent()** and **View.OnTouchListener** used with touch events in old versions of Android.

## Starting a Drag Event

You start with creating a **ClipData** and **ClipData.Item** for the data being moved. As part of the *ClipData* object, supply metadata that is stored in a **ClipDescription** object within the ClipData. For a drag and drop operation that does not represent data movement, you may want to use **null** instead of an actual object.

Next either you can extend extend **View.DragShadowBuilder** to create a drag shadow for dragging the view or simply you can use *View.DragShadowBuilder(View)* to create a default drag shadow that's the same size as the View argument passed to it, with the touch point centered in the drag shadow.

## Example

Following example shows the functionality of a simple Drag & Drop using **View.setOnLongClickListener()**, **View.setOnTouchListener()**and **View.OnDragEventListener()**.

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use Android studio IDE to create an Android application and name it as *My Application* under a package *com.example.saira\_000.myapplication*. |
| 2 | Modify *src/MainActivity.java* file and add the code to define event listeners as well as a call back methods for the logo image used in the example. |
| 3 | Copy image abc.png in *res/drawable-\** folders. You can use images with different resolution in case you want to provide them for different devices. |
| 4 | Modify layout XML file *res/layout/activity\_main.xml* to define default view of the logo images. |
| 5 | 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/MainActivity.java**. This file can include each of the fundamental lifecycle methods.

package com.example.saira\_000.myapplication;

import android.app.Activity;

import android.content.ClipData;

import android.content.ClipDescription;

import android.support.v7.app.ActionBarActivity;

import android.os.Bundle;

import android.util.Log;

import android.view.DragEvent;

import android.view.Menu;

import android.view.MenuItem;

import android.view.MotionEvent;

import android.view.View;

import android.widget.ImageView;

import android.widget.RelativeLayout;

public class MainActivity extends Activity {

ImageView img;

String msg;

private android.widget.RelativeLayout.LayoutParams layoutParams;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

img=(ImageView)findViewById(R.id.imageView);

img.setOnLongClickListener(new View.OnLongClickListener() {

@Override

public boolean onLongClick(View v) {

ClipData.Item item = new ClipData.Item((CharSequence)v.getTag());

String[] mimeTypes = {ClipDescription.MIMETYPE\_TEXT\_PLAIN};

ClipData dragData = new ClipData(v.getTag().toString(),mimeTypes, item);

View.DragShadowBuilder myShadow = new View.DragShadowBuilder(img);

v.startDrag(dragData,myShadow,null,0);

return true;

}

});

img.setOnDragListener(new View.OnDragListener() {

@Override

public boolean onDrag(View v, DragEvent event) {

switch(event.getAction()) {

case DragEvent.ACTION\_DRAG\_STARTED:

layoutParams = (RelativeLayout.LayoutParams)v.getLayoutParams();

Log.d(msg, "Action is DragEvent.ACTION\_DRAG\_STARTED");

// Do nothing

break;

case DragEvent.ACTION\_DRAG\_ENTERED:

Log.d(msg, "Action is DragEvent.ACTION\_DRAG\_ENTERED");

int x\_cord = (int) event.getX();

int y\_cord = (int) event.getY();

break;

case DragEvent.ACTION\_DRAG\_EXITED :

Log.d(msg, "Action is DragEvent.ACTION\_DRAG\_EXITED");

x\_cord = (int) event.getX();

y\_cord = (int) event.getY();

layoutParams.leftMargin = x\_cord;

layoutParams.topMargin = y\_cord;

v.setLayoutParams(layoutParams);

break;

case DragEvent.ACTION\_DRAG\_LOCATION :

Log.d(msg, "Action is DragEvent.ACTION\_DRAG\_LOCATION");

x\_cord = (int) event.getX();

y\_cord = (int) event.getY();

break;

case DragEvent.ACTION\_DRAG\_ENDED :

Log.d(msg, "Action is DragEvent.ACTION\_DRAG\_ENDED");

// Do nothing

break;

case DragEvent.ACTION\_DROP:

Log.d(msg, "ACTION\_DROP event");

// Do nothing

break;

default: break;

}

return true;

}

});

img.setOnTouchListener(new View.OnTouchListener() {

@Override

public boolean onTouch(View v, MotionEvent event) {

if (event.getAction() == MotionEvent.ACTION\_DOWN) {

ClipData data = ClipData.newPlainText("", "");

View.DragShadowBuilder shadowBuilder = new View.DragShadowBuilder(img);

img.startDrag(data, shadowBuilder, img, 0);

img.setVisibility(View.INVISIBLE);

return true;

} else {

return false;

}

}

});

}

}

Following will be the content of **res/layout/activity\_main.xml** file −

In the following code abc indicates the logo of tutorialspoint.com

<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:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Drag and Drop Example"

android:id="@+id/textView"

android:layout\_alignParentTop="true"

android:layout\_centerHorizontal="true"

android:textSize="30dp" />

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Tutorials Point"

android:id="@+id/textView2"

android:layout\_below="@+id/textView"

android:layout\_centerHorizontal="true"

android:textSize="30dp"

android:textColor="#ff14be3c" />>

<ImageView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/imageView"

android:src="@drawable/abc"

android:layout\_below="@+id/textView2"

android:layout\_alignRight="@+id/textView2"

android:layout\_alignEnd="@+id/textView2"

android:layout\_alignLeft="@+id/textView2"

android:layout\_alignStart="@+id/textView2" />

</RelativeLayout>

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

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

<resources>

<string name="app\_name">My Application</string>

</resources>

Following is the default content of **AndroidManifest.xml** −

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

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.saira\_000.myapplication" >

<application

android:allowBackup="true"

android:icon="@drawable/ic\_launcher"

android:label="@string/app\_name"

android:theme="@style/AppTheme" >

<activity

android:name=".MainActivity"

android:label="@string/app\_name" >

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

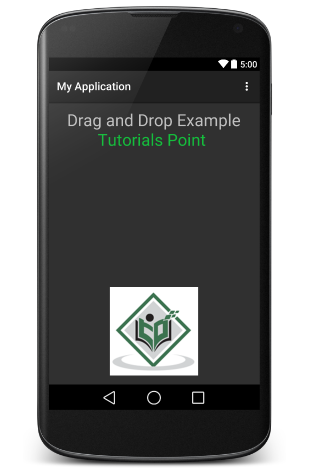
</application>

</manifest>

Let's try to run your **My Application** application. I assume you had created your **AVD** while doing environment setup. To run the app from Android Studio, open one of your project's activity files and click Run icon from the toolbar. Android studio installs the app on your AVD and starts it and if everything is fine with your setup and application, it will display following Emulator window −



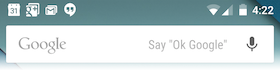
Now do long click on the displayed TutorialsPoint logo and you will see that logo image moves a little after 1 seconds long click from its place, its the time when you should start dragging the image. You can drag it around the screen and drop it at a new location.



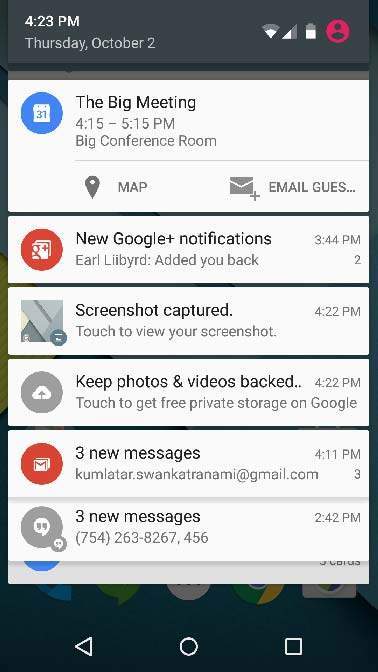
# Android - Notifications

A **notification** is a message you can display to the user outside of your application's normal UI. When you tell the system to issue a notification, it first appears as an icon in the notification area. To see the details of the notification, the user opens the notification drawer. Both the notification area and the notification drawer are system-controlled areas that the user can view at any time.

Android **Toast** class provides a handy way to show users alerts but problem is that these alerts are not persistent which means alert flashes on the screen for a few seconds and then disappears.



To see the details of the notification, you will have to select the icon which will display notification drawer having detail about the notification. While working with emulator with virtual device, you will have to click and drag down the status bar to expand it which will give you detail as follows. This will be just **64 dp** tall and called normal view.



Above expanded form can have a **Big View** which will have additional detail about the notification. You can add upto six additional lines in the notification. The following screen shot shows such notification.

## Create and Send Notifications

You have simple way to create a notification. Follow the following steps in your application to create a notification −

### Step 1 - Create Notification Builder

As a first step is to create a notification builder using *NotificationCompat.Builder.build()*. You will use Notification Builder to set various Notification properties like its small and large icons, title, priority etc.

NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this)

### Step 2 - Setting Notification Properties

Once you have **Builder** object, you can set its Notification properties using Builder object as per your requirement. But this is mandatory to set at least following −

* A small icon, set by **setSmallIcon()**
* A title, set by **setContentTitle()**
* Detail text, set by **setContentText()**

mBuilder.setSmallIcon(R.drawable.notification\_icon);

mBuilder.setContentTitle("Notification Alert, Click Me!");

mBuilder.setContentText("Hi, This is Android Notification Detail!");

You have plenty of optional properties which you can set for your notification. To learn more about them, see the reference documentation for NotificationCompat.Builder.

### Step 3 - Attach Actions

This is an optional part and required if you want to attach an action with the notification. An action allows users to go directly from the notification to an **Activity** in your application, where they can look at one or more events or do further work.

The action is defined by a **PendingIntent** containing an **Intent** that starts an Activity in your application. To associate the PendingIntent with a gesture, call the appropriate method of *NotificationCompat.Builder*. For example, if you want to start Activity when the user clicks the notification text in the notification drawer, you add the PendingIntent by calling **setContentIntent()**.

A PendingIntent object helps you to perform an action on your applications behalf, often at a later time, without caring of whether or not your application is running.

We take help of stack builder object which will contain an artificial back stack for the started Activity. This ensures that navigating backward from the Activity leads out of your application to the Home screen.

Intent resultIntent = new Intent(this, ResultActivity.class);

TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);

stackBuilder.addParentStack(ResultActivity.class);

// Adds the Intent that starts the Activity to the top of the stack

stackBuilder.addNextIntent(resultIntent);

PendingIntent resultPendingIntent = stackBuilder.getPendingIntent(0,PendingIntent.FLAG\_UPDATE\_CURRENT);

mBuilder.setContentIntent(resultPendingIntent);

### Step 4 - Issue the notification

Finally, you pass the Notification object to the system by calling NotificationManager.notify() to send your notification. Make sure you call **NotificationCompat.Builder.build()** method on builder object before notifying it. This method combines all of the options that have been set and return a new **Notification** object.

NotificationManager mNotificationManager = (NotificationManager) getSystemService(Context.NOTIFICATION\_SERVICE);

// notificationID allows you to update the notification later on.

mNotificationManager.notify(notificationID, mBuilder.build());

## The NotificationCompat.Builder Class

The NotificationCompat.Builder class allows easier control over all the flags, as well as help constructing the typical notification layouts. Following are few important and most frequently used methods available as a part of NotificationCompat.Builder class.

|  |  |
| --- | --- |
| **Sr.No.** | **Constants & Description** |
| 1 | **Notification build()**  Combine all of the options that have been set and return a new Notification object. |
| 2 | **NotificationCompat.Builder setAutoCancel (boolean autoCancel)**  Setting this flag will make it so the notification is automatically canceled when the user clicks it in the panel. |
| 3 | **NotificationCompat.Builder setContent (RemoteViews views)**  Supply a custom RemoteViews to use instead of the standard one. |
| 4 | **NotificationCompat.Builder setContentInfo (CharSequence info)**  Set the large text at the right-hand side of the notification. |
| 5 | **NotificationCompat.Builder setContentIntent (PendingIntent intent)**  Supply a PendingIntent to send when the notification is clicked. |
| 6 | **NotificationCompat.Builder setContentText (CharSequence text)**  Set the text (second row) of the notification, in a standard notification. |
| 7 | **NotificationCompat.Builder setContentTitle (CharSequence title)**  Set the text (first row) of the notification, in a standard notification. |
| 8 | **NotificationCompat.Builder setDefaults (int defaults)**  Set the default notification options that will be used. |
| 9 | **NotificationCompat.Builder setLargeIcon (Bitmap icon)**  Set the large icon that is shown in the ticker and notification. |
| 10 | **NotificationCompat.Builder setNumber (int number)**  Set the large number at the right-hand side of the notification. |
| 11 | **NotificationCompat.Builder setOngoing (boolean ongoing)**  Set whether this is an ongoing notification. |
| 12 | **NotificationCompat.Builder setSmallIcon (int icon)**  Set the small icon to use in the notification layouts. |
| 13 | **NotificationCompat.Builder setStyle (NotificationCompat.Style style)**  Add a rich notification style to be applied at build time. |
| 14 | **NotificationCompat.Builder setTicker (CharSequence tickerText)**  Set the text that is displayed in the status bar when the notification first arrives. |
| 15 | **NotificationCompat.Builder setVibrate (long[] pattern)**  Set the vibration pattern to use. |
| 16 | **NotificationCompat.Builder setWhen (long when)**  Set the time that the event occurred. Notifications in the panel are sorted by this time. |

## Example

Following example shows the functionality of a Android notification using a **NotificationCompat.Builder** Class which has been introduced in Android 4.1.

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use Android studio IDE to create an Android application and name it as *tutorialspoint* under a package *com.example.notificationdemo*. |
| 2 | Modify *src/MainActivity.java* file and add the code to notify(""), if user click on the button,it will call android notification service. |
| 3 | Create a new Java file *src/NotificationView.java*, which will be used to display new layout as a part of new activity which will be started when user will click any of the notifications |
| 4 | Modify layout XML file *res/layout/activity\_main.xml* to add Notification button in relative layout. |
| 5 | Create a new layout XML file *res/layout/notification.xml*. This will be used as layout file for new activity which will start when user will click any of the notifications. |
| 6 | No need to change default string constants. Android studio takes care of default string constants |
| 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.notificationdemo/MainActivity.java**. This file can include each of the fundamental lifecycle methods.

package com.example.notificationdemo;

import android.app.Activity;

import android.app.NotificationManager;

import android.app.PendingIntent;

import android.content.Context;

import android.content.Intent;

import android.support.v4.app.NotificationCompat;

import android.os.Bundle;

import android.view.View;

import android.widget.Button;

public class MainActivity extends Activity {

Button b1;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

b1 = (Button)findViewById(R.id.button);

b1.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

addNotification();

}

});

}

private void addNotification() {

NotificationCompat.Builder builder =

new NotificationCompat.Builder(this)

.setSmallIcon(R.drawable.abc)

.setContentTitle("Notifications Example")

.setContentText("This is a test notification");

Intent notificationIntent = new Intent(this, MainActivity.class);

PendingIntent contentIntent = PendingIntent.getActivity(this, 0, notificationIntent,

PendingIntent.FLAG\_UPDATE\_CURRENT);

builder.setContentIntent(contentIntent);

// Add as notification

NotificationManager manager = (NotificationManager) getSystemService(Context.NOTIFICATION\_SERVICE);

manager.notify(0, builder.build());

}

}

Following will be the content of **res/layout/notification.xml** file −

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

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:orientation="vertical"

android:layout\_width="fill\_parent"

android:layout\_height="fill\_parent" >

<TextView

android:layout\_width="fill\_parent"

android:layout\_height="400dp"

android:text="Hi, Your Detailed notification view goes here...." />

</LinearLayout>

Following is the content of the modified main activity file **src/com.example.notificationdemo/NotificationView.java**.

package com.example.notificationdemo;

import android.os.Bundle;

import android.app.Activity;

public class NotificationView extends Activity{

@Override

public void onCreate(Bundle savedInstanceState){

super.onCreate(savedInstanceState);

setContentView(R.layout.notification);

}

}

Following will be the content of **res/layout/activity\_main.xml** file −

<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:paddingBottom="@dimen/activity\_vertical\_margin"

android:paddingLeft="@dimen/activity\_horizontal\_margin"

android:paddingRight="@dimen/activity\_horizontal\_margin"

android:paddingTop="@dimen/activity\_vertical\_margin"

tools:context="MainActivity">

<TextView

android:id="@+id/textView1"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Notification Example"

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\_below="@+id/textView1"

android:layout\_centerHorizontal="true"

android:layout\_marginTop="48dp" />

<ImageButton

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/imageButton"

android:src="@drawable/abc"

android:layout\_below="@+id/textView2"

android:layout\_centerHorizontal="true"

android:layout\_marginTop="42dp" />

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Notification"

android:id="@+id/button"

android:layout\_marginTop="62dp"

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 −

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

<resources>

<string name="action\_settings">Settings</string>

<string name="app\_name">tutorialspoint </string>

</resources>

Following is the default content of **AndroidManifest.xml** −

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

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.notificationdemo" >

<application

android:allowBackup="true"

android:icon="@drawable/ic\_launcher"

android:label="@string/app\_name"

android:theme="@style/AppTheme" >

<activity

android:name="com.example.notificationdemo.MainActivity"

android:label="@string/app\_name" >

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

<activity android:name=".NotificationView"

android:label="Details of notification"

android:parentActivityName=".MainActivity">

<meta-data

android:name="android.support.PARENT\_ACTIVITY"

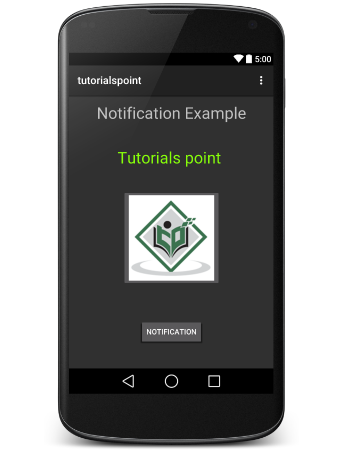
android:value=".MainActivity"/>

</activity>

</application>

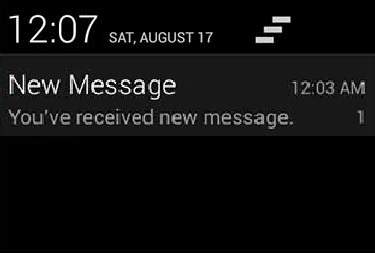
</manifest>

Let's try to run your **tutorialspoint** application. I assume you had created your **AVD** while doing environment set-up. To run the APP from Android Studio, open one of your project's activity files and click Run Eclipse Run Iconicon from the toolbar. Android Studio installs the app on your AVD and starts it and if everything is fine with your setup and application, it will display following Emulator window −



Now click **button**, you will see at the top a message "New Message Alert!" will display momentarily and after that you will have following screen having a small icon at the top left corner.

Now lets expand the view, long click on the small icon, after a second it will display date information and this is the time when you should drag status bar down without releasing mouse. You will see status bar will expand and you will get following screen −



## Big View Notification

The following code snippet demonstrates how to alter the notification created in the previous snippet to use the Inbox big view style. I'm going to update displayNotification() modification method to show this functionality −

protected void displayNotification() {

Log.i("Start", "notification");

/\* Invoking the default notification service \*/

NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this);

mBuilder.setContentTitle("New Message");

mBuilder.setContentText("You've received new message.");

mBuilder.setTicker("New Message Alert!");

mBuilder.setSmallIcon(R.drawable.woman);

/\* Increase notification number every time a new notification arrives \*/

mBuilder.setNumber(++numMessages);

/\* Add Big View Specific Configuration \*/

NotificationCompat.InboxStyle inboxStyle = new NotificationCompat.InboxStyle();

String[] events = new String[6];

events[0] = new String("This is first line....");

events[1] = new String("This is second line...");

events[2] = new String("This is third line...");

events[3] = new String("This is 4th line...");

events[4] = new String("This is 5th line...");

events[5] = new String("This is 6th line...");

// Sets a title for the Inbox style big view

inboxStyle.setBigContentTitle("Big Title Details:");

// Moves events into the big view

for (int i=0; i < events.length; i++) {

inboxStyle.addLine(events[i]);

}

mBuilder.setStyle(inboxStyle);

/\* Creates an explicit intent for an Activity in your app \*/

Intent resultIntent = new Intent(this, NotificationView.class);

TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);

stackBuilder.addParentStack(NotificationView.class);

/\* Adds the Intent that starts the Activity to the top of the stack \*/

stackBuilder.addNextIntent(resultIntent);

PendingIntent resultPendingIntent =stackBuilder.getPendingIntent(0,PendingIntent.FLAG\_UPDATE\_CURRENT);

mBuilder.setContentIntent(resultPendingIntent);

mNotificationManager = (NotificationManager) getSystemService(Context.NOTIFICATION\_SERVICE);

/\* notificationID allows you to update the notification later on. \*/

mNotificationManager.notify(notificationID, mBuilder.build());

}

Now if you will try to run your application then you will find following result in expanded form of the view −



# Android - Location Based Services

Android location APIs make it easy for you to build location-aware applications, without needing to focus on the details of the underlying location technology.

This becomes possible with the help of **Google Play services**, which facilitates adding location awareness to your app with automated location tracking, geofencing, and activity recognition.

This tutorial shows you how to use Location Services in your APP to get the current location, get periodic location updates, look up addresses etc.

## The Location Object

The **Location** object represents a geographic location which can consist of a latitude, longitude, time stamp, and other information such as bearing, altitude and velocity. There are following important methods which you can use with Location object to get location specific information −

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **float distanceTo(Location dest)**  Returns the approximate distance in meters between this location and the given location. |
| 2 | **float getAccuracy()**  Get the estimated accuracy of this location, in meters. |
| 3 | **double getAltitude()**  Get the altitude if available, in meters above sea level. |
| 4 | **float getBearing()**  Get the bearing, in degrees. |
| 5 | **double getLatitude()**  Get the latitude, in degrees. |
| 6 | **double getLongitude()**  Get the longitude, in degrees. |
| 7 | **float getSpeed()**  Get the speed if it is available, in meters/second over ground. |
| 8 | **boolean hasAccuracy()**  True if this location has an accuracy. |
| 9 | **boolean hasAltitude()**  True if this location has an altitude. |
| 10 | **boolean hasBearing()**  True if this location has a bearing. |
| 11 | **boolean hasSpeed()**  True if this location has a speed. |
| 12 | **void reset()**  Clears the contents of the location. |
| 13 | **void setAccuracy(float accuracy)**  Set the estimated accuracy of this location, meters. |
| 14 | **void setAltitude(double altitude)**  Set the altitude, in meters above sea level. |
| 15 | **void setBearing(float bearing)**  Set the bearing, in degrees. |
| 16 | **void setLatitude(double latitude)**  Set the latitude, in degrees. |
| 17 | **void setLongitude(double longitude)**  Set the longitude, in degrees. |
| 18 | **void setSpeed(float speed)**  Set the speed, in meters/second over ground. |
| 19 | **String toString()**  Returns a string containing a concise, human-readable description of this object. |

## Get the Current Location

To get the current location, create a location client which is **LocationClient** object, connect it to Location Services using **connect()** method, and then call its **getLastLocation()** method. This method returns the most recent location in the form of **Location** object that contains latitude and longitude coordinates and other information as explained above. To have location based functionality in your activity, you will have to implement two interfaces −

* GooglePlayServicesClient.ConnectionCallbacks
* GooglePlayServicesClient.OnConnectionFailedListener

These interfaces provide following important callback methods, which you need to implement in your activity class −

|  |  |
| --- | --- |
| **Sr.No.** | **Callback Methods & Description** |
| 1 | **abstract void onConnected(Bundle connectionHint)**  This callback method is called when location service is connected to the location client successfully. You will use **connect()** method to connect to the location client. |
| 2 | **abstract void onDisconnected()**  This callback method is called when the client is disconnected. You will use **disconnect()** method to disconnect from the location client. |
| 3 | **abstract void onConnectionFailed(ConnectionResult result)**  This callback method is called when there was an error connecting the client to the service. |

You should create the location client in **onCreate()** method of your activity class, then connect it in **onStart()**, so that Location Services maintains the current location while your activity is fully visible. You should disconnect the client in **onStop()** method, so that when your app is not visible, Location Services is not maintaining the current location. This helps in saving battery power up-to a large extent.

## Get the Updated Location

If you are willing to have location updates, then apart from above mentioned interfaces, you will need to implement **LocationListener** interface as well. This interface provide following callback method, which you need to implement in your activity class −

|  |  |
| --- | --- |
| **Sr.No.** | **Callback Method & Description** |
| 1 | **abstract void onLocationChanged(Location location)**  This callback method is used for receiving notifications from the LocationClient when the location has changed. |

## Location Quality of Service

The **LocationRequest** object is used to request a quality of service (QoS) for location updates from the **LocationClient**. There are following useful setter methods which you can use to handle QoS. There are equivalent getter methods available which you can check in Android official documentation.

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **setExpirationDuration(long millis)**  Set the duration of this request, in milliseconds. |
| 2 | **setExpirationTime(long millis)**  Set the request expiration time, in millisecond since boot. |
| 3 | **setFastestInterval(long millis)**  Explicitly set the fastest interval for location updates, in milliseconds. |
| 4 | **setInterval(long millis)**  Set the desired interval for active location updates, in milliseconds. |
| 5 | **setNumUpdates(int numUpdates)**  Set the number of location updates. |
| 6 | **setPriority(int priority)**  Set the priority of the request. |

Now for example, if your application wants high accuracy location it should create a location request with **setPriority(int)** set to PRIORITY\_HIGH\_ACCURACY and **setInterval(long)** to 5 seconds. You can also use bigger interval and/or other priorities like PRIORITY\_LOW\_POWER for to request "city" level accuracy or PRIORITY\_BALANCED\_POWER\_ACCURACY for "block" level accuracy.

Activities should strongly consider removing all location request when entering the background (for example at onPause()), or at least swap the request to a larger interval and lower quality to save power consumption.

## Displaying a Location Address

Once you have **Location** object, you can use **Geocoder.getFromLocation()** method to get an address for a given latitude and longitude. This method is synchronous, and may take a long time to do its work, so you should call the method from the **doInBackground()** method of an **AsyncTask** class.

The **AsyncTask** must be subclassed to be used and the subclass will override **doInBackground(Params...)** method to perform a task in the background and **onPostExecute(Result)** method is invoked on the UI thread after the background computation finishes and at the time to display the result. There is one more important method available in AyncTask which is **execute(Params... params)**, this method executes the task with the specified parameters.

## Example

Following example shows you in practical how to to use Location Services in your app to get the current location and its equivalent addresses etc.

To experiment with this example, you will need actual Mobile device equipped with latest Android OS, otherwise you will have to struggle with emulator which may not work.

### Create Android Application

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use Android studio IDE to create an Android application and name it as *Tutorialspoint* under a package *com.example.tutorialspoint7.myapplication*. |
| 2 | add *src/GPSTracker.java* file and add required code. |
| 3 | Modify *src/MainActivity.java* file and add required code as shown below to take care of getting current location and its equivalent address. |
| 4 | Modify layout XML file *res/layout/activity\_main.xml* to add all GUI components which include three buttons and two text views to show location/address. |
| 5 | Modify *res/values/strings.xml* to define required constant values |
| 6 | Modify *AndroidManifest.xml* as shown below |
| 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 **MainActivity.java**.

package com.example.tutorialspoint7.myapplication;

import android.Manifest;

import android.app.Activity;

import android.os.Bundle;

import android.support.v4.app.ActivityCompat;

import android.test.mock.MockPackageManager;

import android.view.View;

import android.widget.Button;

import android.widget.Toast;

public class MainActivity extends Activity {

Button btnShowLocation;

private static final int REQUEST\_CODE\_PERMISSION = 2;

String mPermission = Manifest.permission.ACCESS\_FINE\_LOCATION;

// GPSTracker class

GPSTracker gps;

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

try {

if (ActivityCompat.checkSelfPermission(this, mPermission)

!= MockPackageManager.PERMISSION\_GRANTED) {

ActivityCompat.requestPermissions(this, new String[]{mPermission},

REQUEST\_CODE\_PERMISSION);

// If any permission above not allowed by user, this condition will

execute every time, else your else part will work

}

} catch (Exception e) {

e.printStackTrace();

}

btnShowLocation = (Button) findViewById(R.id.button);

// show location button click event

btnShowLocation.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View arg0) {

// create class object

gps = new GPSTracker(MainActivity.this);

// check if GPS enabled

if(gps.canGetLocation()){

double latitude = gps.getLatitude();

double longitude = gps.getLongitude();

// \n is for new line

Toast.makeText(getApplicationContext(), "Your Location is - \nLat: "

+ latitude + "\nLong: " + longitude, Toast.LENGTH\_LONG).show();

}else{

// can't get location

// GPS or Network is not enabled

// Ask user to enable GPS/network in settings

gps.showSettingsAlert();

}

}

});

}

}

Following is the content of the modified main activity file **GPSTracker.java**.

package com.example.tutorialspoint7.myapplication;

import android.app.AlertDialog;

import android.app.Service;

import android.content.Context;

import android.content.DialogInterface;

import android.content.Intent;

import android.location.Location;

import android.location.LocationListener;

import android.location.LocationManager;

import android.os.Bundle;

import android.os.IBinder;

import android.provider.Settings;

import android.util.Log;

public class GPSTracker extends Service implements LocationListener {

private final Context mContext;

// flag for GPS status

boolean isGPSEnabled = false;

// flag for network status

boolean isNetworkEnabled = false;

// flag for GPS status

boolean canGetLocation = false;

Location location; // location

double latitude; // latitude

double longitude; // longitude

// The minimum distance to change Updates in meters

private static final long MIN\_DISTANCE\_CHANGE\_FOR\_UPDATES = 10; // 10 meters

// The minimum time between updates in milliseconds

private static final long MIN\_TIME\_BW\_UPDATES = 1000 \* 60 \* 1; // 1 minute

// Declaring a Location Manager

protected LocationManager locationManager;

public GPSTracker(Context context) {

this.mContext = context;

getLocation();

}

public Location getLocation() {

try {

locationManager = (LocationManager) mContext.getSystemService(LOCATION\_SERVICE);

// getting GPS status

isGPSEnabled = locationManager.isProviderEnabled(LocationManager.GPS\_PROVIDER);

// getting network status

isNetworkEnabled = locationManager

.isProviderEnabled(LocationManager.NETWORK\_PROVIDER);

if (!isGPSEnabled && !isNetworkEnabled) {

// no network provider is enabled

} else {

this.canGetLocation = true;

// First get location from Network Provider

if (isNetworkEnabled) {

locationManager.requestLocationUpdates(

LocationManager.NETWORK\_PROVIDER,

MIN\_TIME\_BW\_UPDATES,

MIN\_DISTANCE\_CHANGE\_FOR\_UPDATES, this);

Log.d("Network", "Network");

if (locationManager != null) {

location = locationManager

.getLastKnownLocation(LocationManager.NETWORK\_PROVIDER);

if (location != null) {

latitude = location.getLatitude();

longitude = location.getLongitude();

}

}

}

// if GPS Enabled get lat/long using GPS Services

if (isGPSEnabled) {

if (location == null) {

locationManager.requestLocationUpdates(

LocationManager.GPS\_PROVIDER,

MIN\_TIME\_BW\_UPDATES,

MIN\_DISTANCE\_CHANGE\_FOR\_UPDATES, this);

Log.d("GPS Enabled", "GPS Enabled");

if (locationManager != null) {

location = locationManager

.getLastKnownLocation(LocationManager.GPS\_PROVIDER);

if (location != null) {

latitude = location.getLatitude();

longitude = location.getLongitude();

}

}

}

}

}

} catch (Exception e) {

e.printStackTrace();

}

return location;

}

/\*\*

\* Stop using GPS listener

\* Calling this function will stop using GPS in your app

\* \*/

public void stopUsingGPS(){

if(locationManager != null){

locationManager.removeUpdates(GPSTracker.this);

}

}

/\*\*

\* Function to get latitude

\* \*/

public double getLatitude(){

if(location != null){

latitude = location.getLatitude();

}

// return latitude

return latitude;

}

/\*\*

\* Function to get longitude

\* \*/

public double getLongitude(){

if(location != null){

longitude = location.getLongitude();

}

// return longitude

return longitude;

}

/\*\*

\* Function to check GPS/wifi enabled

\* @return boolean

\* \*/

public boolean canGetLocation() {

return this.canGetLocation;

}

/\*\*

\* Function to show settings alert dialog

\* On pressing Settings button will lauch Settings Options

\* \*/

public void showSettingsAlert(){

AlertDialog.Builder alertDialog = new AlertDialog.Builder(mContext);

// Setting Dialog Title

alertDialog.setTitle("GPS is settings");

// Setting Dialog Message

alertDialog.setMessage("GPS is not enabled. Do you want to go to settings menu?");

// On pressing Settings button

alertDialog.setPositiveButton("Settings", new DialogInterface.OnClickListener() {

public void onClick(DialogInterface dialog,int which) {

Intent intent = new Intent(Settings.ACTION\_LOCATION\_SOURCE\_SETTINGS);

mContext.startActivity(intent);

}

});

// on pressing cancel button

alertDialog.setNegativeButton("Cancel", new DialogInterface.OnClickListener() {

public void onClick(DialogInterface dialog, int which) {

dialog.cancel();

}

});

// Showing Alert Message

alertDialog.show();

}

@Override

public void onLocationChanged(Location location) {

}

@Override

public void onProviderDisabled(String provider) {

}

@Override

public void onProviderEnabled(String provider) {

}

@Override

public void onStatusChanged(String provider, int status, Bundle extras) {

}

@Override

public IBinder onBind(Intent arg0) {

return null;

}

}

Following will be the content of **res/layout/activity\_main.xml** file −

<?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" >

<Button

android:id = "@+id/button"

android:layout\_width = "fill\_parent"

android:layout\_height = "wrap\_content"

android:text = "getlocation"/>

</LinearLayout>

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

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

<resources>

<string name = "app\_name">Tutorialspoint</string>

</resources>

Following is the default content of **AndroidManifest.xml** −

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

<manifest xmlns:android = "http://schemas.android.com/apk/res/android"

package = "com.example.tutorialspoint7.myapplication">

<uses-permission android:name = "android.permission.ACCESS\_FINE\_LOCATION" />

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

<application

android:allowBackup = "true"

android:icon = "@mipmap/ic\_launcher"

android:label = "@string/app\_name"

android:supportsRtl = "true"

android:theme = "@style/AppTheme">

<activity android:name = ".MainActivity">

<intent-filter>

<action android:name = "android.intent.action.MAIN" />

<category android:name = "android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

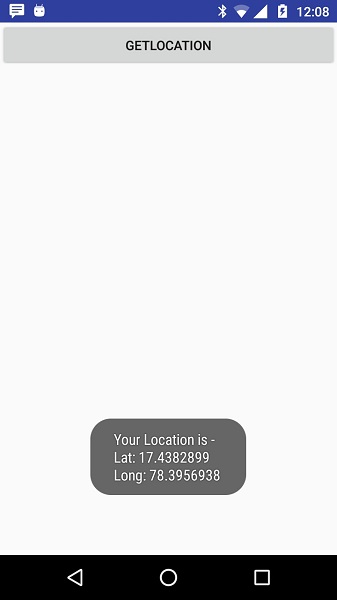
</application>

</manifest>

Let's try to run your **Tutorialspoint** application. I assume that, 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 Eclipse Run Iconicon from the toolbar. Before starting your application, Android studio installer will display following window to select an option where you want to run your Android application.



Now to see location select Get Location Button which will display location information as follows −



# Android - Sending Email

**Email** is messages distributed by electronic means from one system user to one or more recipients via a network.

Before starting Email Activity, You must know Email functionality with intent, Intent is carrying data from one component to another component with-in the application or outside the application.

To send an email from your application, you don’t have to implement an email client from the beginning, but you can use an existing one like the default Email app provided from Android, Gmail, Outlook, K-9 Mail etc. For this purpose, we need to write an Activity that launches an email client, using an implicit Intent with the right action and data. In this example, we are going to send an email from our app by using an Intent object that launches existing email clients.

Following section explains different parts of our Intent object required to send an email.

## Intent Object - Action to send Email

You will use **ACTION\_SEND** action to launch an email client installed on your Android device. Following is simple syntax to create an intent with ACTION\_SEND action.

Intent emailIntent = new Intent(Intent.ACTION\_SEND);

## Intent Object - Data/Type to send Email

To send an email you need to specify **mailto:** as URI using setData() method and data type will be to **text/plain** using setType() method as follows −

emailIntent.setData(Uri.parse("mailto:"));

emailIntent.setType("text/plain");

## Intent Object - Extra to send Email

Android has built-in support to add TO, SUBJECT, CC, TEXT etc. fields which can be attached to the intent before sending the intent to a target email client. You can use following extra fields in your email −

|  |  |
| --- | --- |
| **Sr.No.** | **Extra Data & Description** |
| 1 | **EXTRA\_BCC**  A String[] holding e-mail addresses that should be blind carbon copied. |
| 2 | **EXTRA\_CC**  A String[] holding e-mail addresses that should be carbon copied. |
| 3 | **EXTRA\_EMAIL**  A String[] holding e-mail addresses that should be delivered to. |
| 4 | **EXTRA\_HTML\_TEXT**  A constant String that is associated with the Intent, used with ACTION\_SEND to supply an alternative to EXTRA\_TEXT as HTML formatted text. |
| 5 | **EXTRA\_SUBJECT**  A constant string holding the desired subject line of a message. |
| 6 | **EXTRA\_TEXT**  A constant CharSequence that is associated with the Intent, used with ACTION\_SEND to supply the literal data to be sent. |
| 7 | **EXTRA\_TITLE**  A CharSequence dialog title to provide to the user when used with a ACTION\_CHOOSER. |

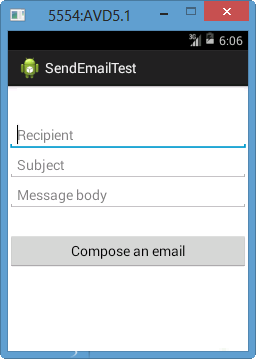
Here is an example showing you how to assign extra data to your intent −

emailIntent.putExtra(Intent.EXTRA\_EMAIL , new String[]{"Recipient"});

emailIntent.putExtra(Intent.EXTRA\_SUBJECT, "subject");

emailIntent.putExtra(Intent.EXTRA\_TEXT , "Message Body");

The out-put of above code is as below shown an image



#### Email Example

## Example

Following example shows you in practical how to use Intent object to launch Email client to send an Email to the given recipients.

To Email experiment with this example, you will need actual Mobile device equipped with latest Android OS, otherwise you might get struggle with emulator which may not work properly. Second you will need to have an Email client like GMail(By default every android version having Gmail client App) or K9mail installed on your device.

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use Android studio to create an Android application and name it as *Tutorialspoint* under a package *com.example.tutorialspoint*. |
| 2 | Modify *src/MainActivity.java* file and add required code to take care of sending email. |
| 3 | Modify layout XML file *res/layout/activity\_main.xml* add any GUI component if required. I'm adding a simple button to launch Email Client. |
| 4 | Modify *res/values/strings.xml* to define required constant values |
| 5 | Modify *AndroidManifest.xml* as shown below |
| 6 | 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.Tutorialspoint/MainActivity.java**.

package com.example.tutorialspoint;

import android.net.Uri;

import android.os.Bundle;

import android.app.Activity;

import android.content.Intent;

import android.util.Log;

import android.view.Menu;

import android.view.View;

import android.widget.Button;

import android.widget.Toast;

public class MainActivity extends Activity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

Button startBtn = (Button) findViewById(R.id.sendEmail);

startBtn.setOnClickListener(new View.OnClickListener() {

public void onClick(View view) {

sendEmail();

}

});

}

protected void sendEmail() {

Log.i("Send email", "");

String[] TO = {""};

String[] CC = {""};

Intent emailIntent = new Intent(Intent.ACTION\_SEND);

emailIntent.setData(Uri.parse("mailto:"));

emailIntent.setType("text/plain");

emailIntent.putExtra(Intent.EXTRA\_EMAIL, TO);

emailIntent.putExtra(Intent.EXTRA\_CC, CC);

emailIntent.putExtra(Intent.EXTRA\_SUBJECT, "Your subject");

emailIntent.putExtra(Intent.EXTRA\_TEXT, "Email message goes here");

try {

startActivity(Intent.createChooser(emailIntent, "Send mail..."));

finish();

Log.i("Finished sending email...", "");

} catch (android.content.ActivityNotFoundException ex) {

Toast.makeText(MainActivity.this, "There is no email client installed.", Toast.LENGTH\_SHORT).show();

}

}

}

Following will be the content of **res/layout/activity\_main.xml** file −

Here abc indicates about tutorialspoint logo

<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/textView1"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Sending Mail Example"

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\_alignRight="@+id/imageButton"

android:layout\_alignEnd="@+id/imageButton" />

<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:id="@+id/sendEmail"

android:layout\_width="fill\_parent"

android:layout\_height="wrap\_content"

android:text="@string/compose\_email"/>

</LinearLayout>

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

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

<resources>

<string name="app\_name">Tutorialspoint</string>

<string name="compose\_email">Compose Email</string>

</resources>

Following is the default content of **AndroidManifest.xml** −

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

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.Tutorialspoint" >

<application

android:allowBackup="true"

android:icon="@drawable/ic\_launcher"

android:label="@string/app\_name"

android:theme="@style/AppTheme" >

<activity

android:name="com.example.tutorialspoint.MainActivity"

android:label="@string/app\_name" >

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

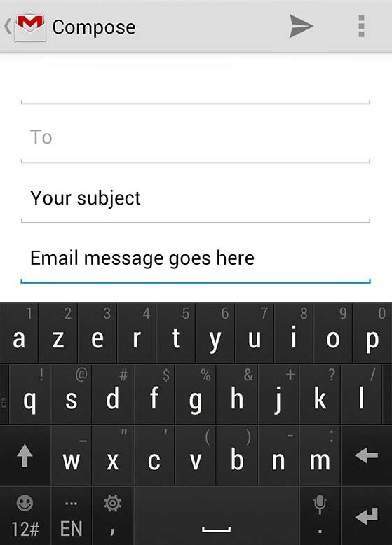
</application>

</manifest>

Let's try to run your **tutorialspoint** 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 Eclipse Run Iconicon from the toolbar. Before starting your application, Android studio installer will display following window to select an option where you want to run your Android application.Select your mobile device as an option and then check your mobile device which will display following screen −



Now use **Compose Email** button to list down all the installed email clients. From the list, you can choose one of email clients to send your email. I'm going to use Gmail client to send my email which will have all the provided defaults fields available as shown below. Here **From:** will be default email ID you have registered for your Android device.



You can modify either of the given default fields and finally use send email button to send your email to the mentioned recipients.

# Android - Sending SMS

In Android, you can use SmsManager API or devices Built-in SMS application to send SMS's. In this tutorial, we shows you two basic examples to send SMS message −

**SmsManager API**

SmsManager smsManager = SmsManager.getDefault();

smsManager.sendTextMessage("phoneNo", null, "sms message", null, null);

**Built-in SMS application**

Intent sendIntent = new Intent(Intent.ACTION\_VIEW);

sendIntent.putExtra("sms\_body", "default content");

sendIntent.setType("vnd.android-dir/mms-sms");

startActivity(sendIntent);

Of course, both need **SEND\_SMS permission**.

<uses-permission android:name="android.permission.SEND\_SMS" />

Apart from the above method, there are few other important functions available in SmsManager class. These methods are listed below −

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **ArrayList<String> divideMessage(String text)**  This method divides a message text into several fragments, none bigger than the maximum SMS message size. |
| 2 | **static SmsManager getDefault()**  This method is used to get the default instance of the SmsManager |
| 3 | **void sendDataMessage(String destinationAddress, String scAddress, short destinationPort, byte[] data, PendingIntent sentIntent, PendingIntent deliveryIntent)**  This method is used to send a data based SMS to a specific application port. |
| 4 | **void sendMultipartTextMessage(String destinationAddress, String scAddress, ArrayList<String> parts, ArrayList<PendingIntent> sentIntents, ArrayList<PendingIntent> deliveryIntents)**  Send a multi-part text based SMS. |
| 5 | **void sendTextMessage(String destinationAddress, String scAddress, String text, PendingIntent sentIntent, PendingIntent deliveryIntent)**  Send a text based SMS. |

**Example**

Following example shows you in practical how to use SmsManager object to send an SMS to the given mobile number.

To experiment with this example, you will need actual Mobile device equipped with latest Android OS, otherwise you will have to struggle with emulator which may not work.

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use Android Studio IDE to create an Android application and name it as *tutorialspoint* under a package *com.example.tutorialspoint*. |
| 2 | Modify *src/MainActivity.java* file and add required code to take care of sending sms. |
| 3 | Modify layout XML file *res/layout/activity\_main.xml* add any GUI component if required. I'm adding a simple GUI to take mobile number and SMS text to be sent and a simple button to send SMS. |
| 4 | No need to define default string constants at res/values/strings.xml. Android studio takes care of default constants. |
| 5 | Modify *AndroidManifest.xml* as shown below |
| 6 | 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.tutorialspoint/MainActivity.java**.

package com.example.tutorialspoint;

import android.Manifest;

import android.content.pm.PackageManager;

import android.os.Bundle;

import android.app.Activity;

import android.support.v4.app.ActivityCompat;

import android.support.v4.content.ContextCompat;

import android.telephony.SmsManager;

import android.util.Log;

import android.view.Menu;

import android.view.View;

import android.widget.Button;

import android.widget.EditText;

import android.widget.Toast;

public class MainActivity extends Activity {

private static final int MY\_PERMISSIONS\_REQUEST\_SEND\_SMS =0 ;

Button sendBtn;

EditText txtphoneNo;

EditText txtMessage;

String phoneNo;

String message;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

sendBtn = (Button) findViewById(R.id.btnSendSMS);

txtphoneNo = (EditText) findViewById(R.id.editText);

txtMessage = (EditText) findViewById(R.id.editText2);

sendBtn.setOnClickListener(new View.OnClickListener() {

public void onClick(View view) {

sendSMSMessage();

}

});

}

protected void sendSMSMessage() {

phoneNo = txtphoneNo.getText().toString();

message = txtMessage.getText().toString();

if (ContextCompat.checkSelfPermission(this,

Manifest.permission.SEND\_SMS)

!= PackageManager.PERMISSION\_GRANTED) {

if (ActivityCompat.shouldShowRequestPermissionRationale(this,

Manifest.permission.SEND\_SMS)) {

} else {

ActivityCompat.requestPermissions(this,

new String[]{Manifest.permission.SEND\_SMS},

MY\_PERMISSIONS\_REQUEST\_SEND\_SMS);

}

}

}

@Override

public void onRequestPermissionsResult(int requestCode,String permissions[], int[] grantResults) {

switch (requestCode) {

case MY\_PERMISSIONS\_REQUEST\_SEND\_SMS: {

if (grantResults.length > 0

&& grantResults[0] == PackageManager.PERMISSION\_GRANTED) {

SmsManager smsManager = SmsManager.getDefault();

smsManager.sendTextMessage(phoneNo, null, message, null, null);

Toast.makeText(getApplicationContext(), "SMS sent.",

Toast.LENGTH\_LONG).show();

} else {

Toast.makeText(getApplicationContext(),

"SMS faild, please try again.", Toast.LENGTH\_LONG).show();

return;

}

}

}

}

}

Following will be the content of **res/layout/activity\_main.xml** file −

Here abc indicates about tutorialspoint logo

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

<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:paddingBottom="@dimen/activity\_vertical\_margin"

android:paddingLeft="@dimen/activity\_horizontal\_margin"

android:paddingRight="@dimen/activity\_horizontal\_margin"

android:paddingTop="@dimen/activity\_vertical\_margin"

tools:context="MainActivity">

<TextView

android:id="@+id/textView1"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Sending SMS Example"

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\_below="@+id/textView1"

android:layout\_alignRight="@+id/imageButton"

android:layout\_alignEnd="@+id/imageButton" />

<ImageButton

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/imageButton"

android:src="@drawable/abc"

android:layout\_below="@+id/textView2"

android:layout\_centerHorizontal="true" />

<EditText

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/editText"

android:hint="Enter Phone Number"

android:phoneNumber="true"

android:textColorHint="@color/abc\_primary\_text\_material\_dark"

android:layout\_below="@+id/imageButton"

android:layout\_centerHorizontal="true" />

<EditText

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/editText2"

android:layout\_below="@+id/editText"

android:layout\_alignLeft="@+id/editText"

android:layout\_alignStart="@+id/editText"

android:textColorHint="@color/abc\_primary\_text\_material\_dark"

android:layout\_alignRight="@+id/imageButton"

android:layout\_alignEnd="@+id/imageButton"

android:hint="Enter SMS" />

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Send Sms"

android:id="@+id/btnSendSMS"

android:layout\_below="@+id/editText2"

android:layout\_centerHorizontal="true"

android:layout\_marginTop="48dp" />

</RelativeLayout>

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

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

<resources>

<string name="app\_name">tutorialspoint</string>

</resources>

Following is the default content of **AndroidManifest.xml** −

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

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.tutorialspoint" >

<uses-permission android:name="android.permission.SEND\_SMS" />

<application

android:allowBackup="true"

android:icon="@drawable/ic\_launcher"

android:label="@string/app\_name"

android:theme="@style/AppTheme" >

<activity

android:name="com.example.tutorialspoint.MainActivity"

android:label="@string/app\_name" >

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

</application>

</manifest>

Let's try to run your **tutorialspoint** 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 Eclipse Run Iconicon from the toolbar. Before starting your application, Android studio installer will display following window to select an option where you want to run your Android application.



Now you can enter a desired mobile number and a text message to be sent on that number. Finally click on **Send SMS** button to send your SMS. Make sure your GSM/CDMA connection is working fine to deliver your SMS to its recipient.

You can take a number of SMS separated by comma and then inside your program you will have to parse them into an array string and finally you can use a loop to send message to all the given numbers. That's how you can write your own SMS client. Next section will show you how to use existing SMS client to send SMS.

**Using Built-in Intent to send SMS**

You can use Android Intent to send SMS by calling built-in SMS functionality of the Android. Following section explains different parts of our Intent object required to send an SMS.

**Intent Object - Action to send SMS**

You will use **ACTION\_VIEW** action to launch an SMS client installed on your Android device. Following is simple syntax to create an intent with ACTION\_VIEW action.

Intent smsIntent = new Intent(Intent.ACTION\_VIEW);

**Intent Object - Data/Type to send SMS**

To send an SMS you need to specify **smsto:** as URI using setData() method and data type will be to **vnd.android-dir/mms-sms** using setType() method as follows −

smsIntent.setData(Uri.parse("smsto:"));

smsIntent.setType("vnd.android-dir/mms-sms");

**Intent Object - Extra to send SMS**

Android has built-in support to add phone number and text message to send an SMS as follows −

smsIntent.putExtra("address" , new String("0123456789;3393993300"));

smsIntent.putExtra("sms\_body" , "Test SMS to Angilla");

Here address and sms\_body are case sensitive and should be specified in small characters only. You can specify more than one number in single string but separated by semi-colon (;).

**Example**

Following example shows you in practical how to use Intent object to launch SMS client to send an SMS to the given recipients.

To experiment with this example, you will need actual Mobile device equipped with latest Android OS, otherwise you will have to struggle with emulator which may not work.

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use Android studio IDE to create an Android application and name it as *tutorialspoint* under a package *com.example.tutorialspoint*. |
| 2 | Modify *src/MainActivity.java* file and add required code to take care of sending SMS. |
| 3 | Modify layout XML file *res/layout/activity\_main.xml* add any GUI component if required. I'm adding a simple button to launch SMS Client. |
| 4 | No need to define default constants.Android studio takes care of default constants. |
| 5 | Modify *AndroidManifest.xml* as shown below |
| 6 | 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.tutorialspoint/MainActivity.java**.

package com.example.tutorialspoint;

import android.net.Uri;

import android.os.Bundle;

import android.app.Activity;

import android.content.Intent;

import android.util.Log;

import android.view.Menu;

import android.view.View;

import android.widget.Button;

import android.widget.Toast;

public class MainActivity extends Activity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

Button startBtn = (Button) findViewById(R.id.button);

startBtn.setOnClickListener(new View.OnClickListener() {

public void onClick(View view) {

sendSMS();

}

});

}

protected void sendSMS() {

Log.i("Send SMS", "");

Intent smsIntent = new Intent(Intent.ACTION\_VIEW);

smsIntent.setData(Uri.parse("smsto:"));

smsIntent.setType("vnd.android-dir/mms-sms");

smsIntent.putExtra("address" , new String ("01234"));

smsIntent.putExtra("sms\_body" , "Test ");

try {

startActivity(smsIntent);

finish();

Log.i("Finished sending SMS...", "");

} catch (android.content.ActivityNotFoundException ex) {

Toast.makeText(MainActivity.this,

"SMS faild, please try again later.", Toast.LENGTH\_SHORT).show();

}

}

@Override

public boolean onCreateOptionsMenu(Menu menu) {

// Inflate the menu; this adds items to the action bar if it is present.

getMenuInflater().inflate(R.menu.main, menu);

return true;

}

}

Following will be the content of **res/layout/activity\_main.xml** file −

Here abc indicates about tutorialspoint logo

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

<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:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Drag and Drop Example"

android:id="@+id/textView"

android:layout\_alignParentTop="true"

android:layout\_centerHorizontal="true"

android:textSize="30dp" />

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Tutorials Point "

android:id="@+id/textView2"

android:layout\_below="@+id/textView"

android:layout\_centerHorizontal="true"

android:textSize="30dp"

android:textColor="#ff14be3c" />

<ImageView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/imageView"

android:src="@drawable/abc"

android:layout\_marginTop="48dp"

android:layout\_below="@+id/textView2"

android:layout\_centerHorizontal="true" />

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Compose SMS"

android:id="@+id/button"

android:layout\_below="@+id/imageView"

android:layout\_alignRight="@+id/textView2"

android:layout\_alignEnd="@+id/textView2"

android:layout\_marginTop="54dp"

android:layout\_alignLeft="@+id/imageView"

android:layout\_alignStart="@+id/imageView" />

</RelativeLayout>

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

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

<resources>

<string name="app\_name">tutorialspoint</string>

</resources>

Following is the default content of **AndroidManifest.xml** −

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

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.tutorialspoint" >

<application

android:allowBackup="true"

android:icon="@drawable/ic\_launcher"

android:label="@string/app\_name"

android:theme="@style/AppTheme" >

<activity

android:name="com.example.tutorialspoint.MainActivity"

android:label="@string/app\_name" >

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

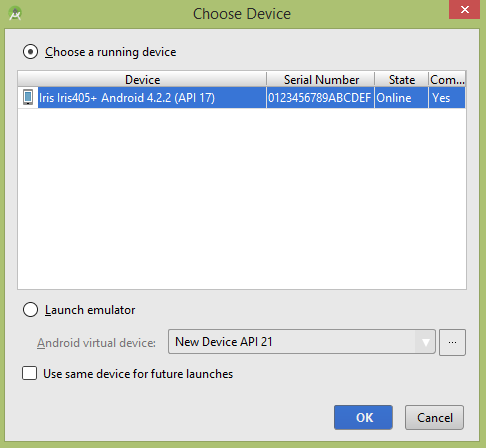
</intent-filter>

</activity>

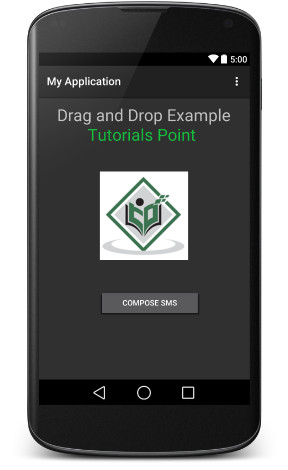
</application>

</manifest>

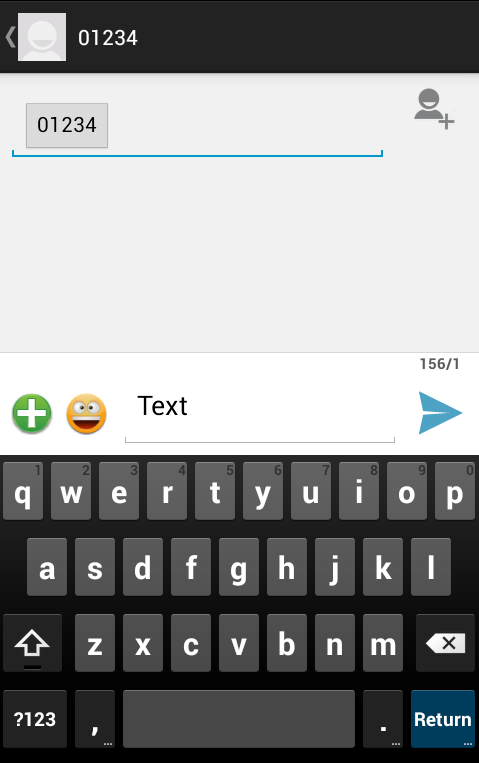
Let's try to run your **tutorialspoint** application. I assume you have connected your actual Android Mobile device with your computer. To run the app from Android studio, open one of your project's activity files and click Run Eclipse Run Iconicon from the toolbar. Before starting your application, Android studio will display following window to select an option where you want to run your Android application.



Select your mobile device as an option and then check your mobile device which will display following screen −



Now use **Compose SMS** button to launch Android built-in SMS clients which is shown below −



You can modify either of the given default fields and finally use send SMS button to send your SMS to the mentioned recipient.

# Android - Phone Calls

Android provides Built-in applications for phone calls, in some occasions we may need to make a phone call through our application. This could easily be done by using implicit Intent with appropriate actions. Also, we can use PhoneStateListener and TelephonyManager classes, in order to monitor the changes in some telephony states on the device.

This chapter lists down all the simple steps to create an application which can be used to make a Phone Call. You can use Android Intent to make phone call by calling built-in Phone Call functionality of the Android. Following section explains different parts of our Intent object required to make a call.

**Intent Object - Action to make Phone Call**

You will use **ACTION\_CALL** action to trigger built-in phone call functionality available in Android device. Following is simple syntax to create an intent with ACTION\_CALL action

Intent phoneIntent = new Intent(Intent.ACTION\_CALL);

You can use **ACTION\_DIAL** action instead of ACTION\_CALL, in that case you will have option to modify hardcoded phone number before making a call instead of making a direct call.

**Intent Object - Data/Type to make Phone Call**

To make a phone call at a given number 91-000-000-0000, you need to specify **tel:** as URI using setData() method as follows −

phoneIntent.setData(Uri.parse("tel:91-000-000-0000"));

The interesting point is that, to make a phone call, you do not need to specify any extra data or data type.

**Example**

Following example shows you in practical how to use Android Intent to make phone call to the given mobile number.

To experiment with this example, you will need actual Mobile device equipped with latest Android OS, otherwise you will have to struggle with emulator which may not work.

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use Android studio IDE to create an Android application and name it as *My Application* under a package *com.example.saira\_000.myapplication*. |
| 2 | Modify *src/MainActivity.java* file and add required code to take care of making a call. |
| 3 | Modify layout XML file *res/layout/activity\_main.xml* add any GUI component if required. I'm adding a simple button to Call 91-000-000-0000 number |
| 4 | No need to define default string constants.Android studio takes care of default constants. |
| 5 | Modify *AndroidManifest.xml* as shown below |
| 6 | 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/MainActivity.java**.

package com.example.saira\_000.myapplication;

import android.Manifest;

import android.content.Intent;

import android.content.pm.PackageManager;

import android.net.Uri;

import android.os.Bundle;

import android.support.v4.app.ActivityCompat;

import android.support.v7.app.AppCompatActivity;

import android.view.View;

import android.widget.Button;

public class MainActivity extends AppCompatActivity {

private Button button;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

button = (Button) findViewById(R.id.buttonCall);

button.setOnClickListener(new View.OnClickListener() {

public void onClick(View arg0) {

Intent callIntent = new Intent(Intent.ACTION\_CALL);

callIntent.setData(Uri.parse("tel:0377778888"));

if (ActivityCompat.checkSelfPermission(MainActivity.this,

Manifest.permission.CALL\_PHONE) != PackageManager.PERMISSION\_GRANTED) {

return;

}

startActivity(callIntent);

}

});

}

}

Following will be the content of **res/layout/activity\_main.xml** file −

<?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" >

<Button

android:id="@+id/buttonCall"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="call 0377778888" />

</LinearLayout>

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

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

<resources>

<string name="app\_name">My Application</string>

</resources>

Following is the default content of **AndroidManifest.xml** −

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

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.saira\_000.myapplication" >

<uses-permission android:name="android.permission.CALL\_PHONE" />

<application

android:allowBackup="true"

android:icon="@drawable/ic\_launcher"

android:label="@string/app\_name"

android:theme="@style/AppTheme" >

<activity

android:name="com.example.saira\_000.myapplication.MainActivity"

android:label="@string/app\_name" >

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

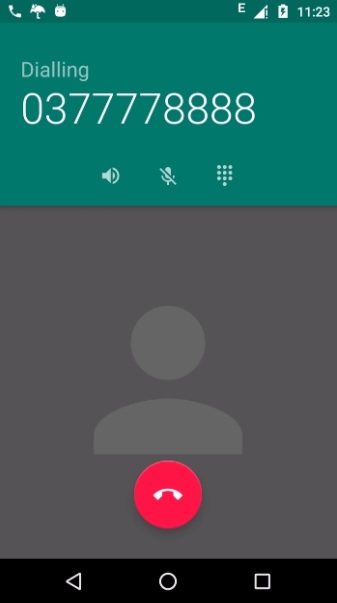
</application>

</manifest>

Let's try to run your **My Application** 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 Eclipse Run Iconicon from the toolbar.Select your mobile device as an option and then check your mobile device which will display following screen −

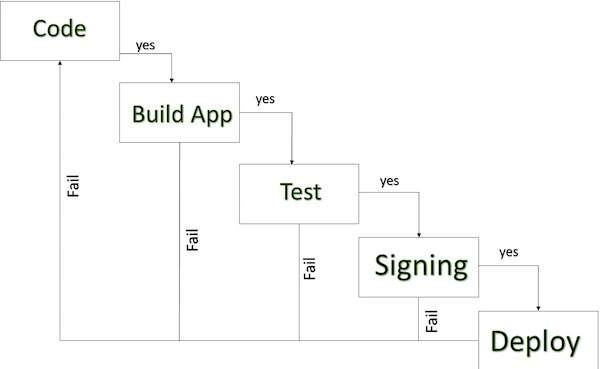


Now use **Call** button to make phone call as shown below −



# Publishing Android Application

Android application publishing is a process that makes your Android applications available to users. Infect, publishing is the last phase of the Android application development process.



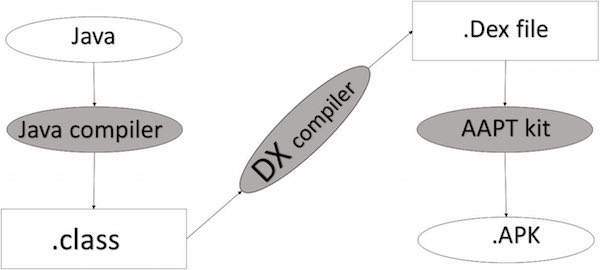
#### Android development life cycle

Once you developed and fully tested your Android Application, you can start selling or distributing free using Google Play (A famous Android marketplace). You can also release your applications by sending them directly to users or by letting users download them from your own website.

You can check a detailed publishing process at Android official website, but this tutorial will take you through simple steps to launch your application on Google Play. Here is a simplified check list which will help you in launching your Android application −

|  |  |
| --- | --- |
| **Step** | **Activity** |
| 1 | **Regression Testing** Before you publish your application, you need to make sure that its meeting the basic quality expectations for all Android apps, on all of the devices that you are targeting. So perform all the required testing on different devices including phone and tablets. |
| 2 | **Application Rating** When you will publish your application at Google Play, you will have to specify a content rating for your app, which informs Google Play users of its maturity level. Currently available ratings are (a) Everyone (b) Low maturity (c) Medium maturity (d) High maturity. |
| 3 | **Targeted Regions** Google Play lets you control what countries and territories where your application will be sold. Accordingly you must take care of setting up time zone, localization or any other specific requirement as per the targeted region. |
| 4 | **Application Size** Currently, the maximum size for an APK published on Google Play is 50 MB. If your app exceeds that size, or if you want to offer a secondary download, you can use APK Expansion Files, which Google Play will host for free on its server infrastructure and automatically handle the download to devices. |
| 5 | **SDK and Screen Compatibility** It is important to make sure that your app is designed to run properly on the Android platform versions and device screen sizes that you want to target. |
| 6 | **Application Pricing** Deciding whether you app will be free or paid is important because, on Google Play, free app's must remain free. If you want to sell your application then you will have to specify its price in different currencies. |
| 7 | **Promotional Content** It is a good marketing practice to supply a variety of high-quality graphic assets to showcase your app or brand. After you publish, these appear on your product details page, in store listings and search results, and elsewhere. |
| 8 | **Build and Upload release-ready APK** The release-ready APK is what you you will upload to the Developer Console and distribute to users. You can check complete detail on how to create a release-ready version of your app: [Preparing for Release](https://developer.android.com/tools/publishing/preparing.html). |
| 9 | **Finalize Application Detail** Google Play gives you a variety of ways to promote your app and engage with users on your product details page, from colourful graphics, screen shots, and videos to localized descriptions, release details, and links to your other apps. So you can decorate your application page and provide as much as clear crisp detail you can provide. |

## Export Android Application Process



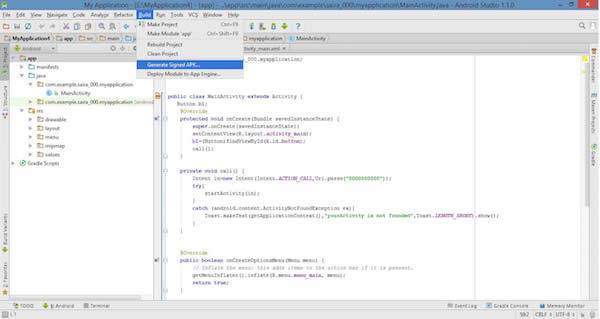
#### Apk development process

Before exporting the apps, you must some of tools

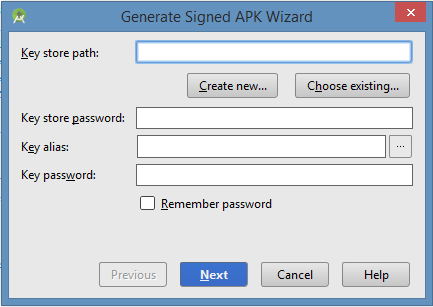
* **Dx tools**(Dalvik executable tools ): It going to convert **.class file** to **.dex file**. it has useful for memory optimization and reduce the boot-up speed time
* **AAPT**(Android assistance packaging tool):it has useful to convert **.Dex file** to**.Apk**
* **APK**(Android packaging kit): The final stage of deployment process is called as .apk.

You will need to export your application as an APK (Android Package) file before you upload it Google Play marketplace.

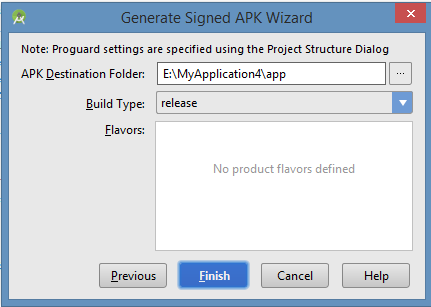
To export an application, just open that application project in Android studio and select **Build → Generate Signed APK** from your Android studio and follow the simple steps to export your application −



Next select, **Generate Signed APK** option as shown in the above screen shot and then click it so that you get following screen where you will choose **Create new keystore** to store your application.



Enter your key store path,key store password,key alias and key password to protect your application and click on **Next** button once again. It will display following screen to let you create an application −



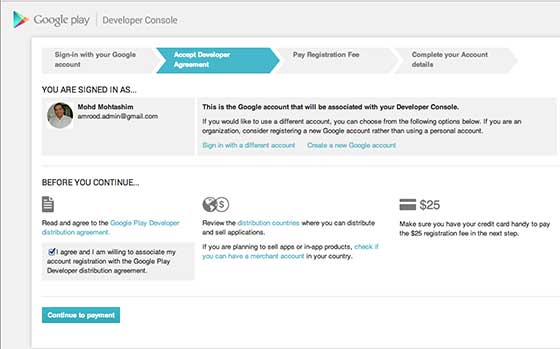
Once you filled up all the information,like app destination,build type and flavours click **finish** button While creating an application it will show as below

Creating Application

Finally, it will generate your Android Application as APK formate File which will be uploaded at Google Play marketplace.

## Google Play Registration

The most important step is to register with Google Play using [Google Play Marketplace](https://play.google.com/apps/publish/). You can use your existing google ID if you have any otherwise you can create a new Google ID and then register with the marketplace. You will have following screen to accept terms and condition.



You can use **Continue to payment** button to proceed to make a payment of $25 as a registration fee and finally to complete your account detail.

Once you are a registered user at Google Play, you can upload **release-ready APK** for your application and finally you will complete application detail using application detail page as mentioned in step 9 of the above mentioned checklist.

## Signing Your App Manually

You do not need Android Studio to sign your app. You can sign your app from the command line using standard tools from the Android SDK and the JDK. To sign an app in release mode from the command line −

* Generate a private key using keytool

$ keytool -genkey -v -keystore my-release-key.keystore

-alias alias\_name -keyalg RSA -keysize 2048 -validity 10000

* Compile your app in release mode to obtain an unsigned APK
* Sign your app with your private key using [jarsigner](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/jarsigner.html)

$ jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1

-keystore my-release-key.keystore my\_application.apk alias\_name

* Verify that your APK is signed. For example −

$ jarsigner -verify -verbose -certs my\_application.apk

* Align the final APK package using [zipalign.](https://developer.android.com/tools/help/zipalign.html)

$ zipalign -v 4 your\_project\_name-unaligned.apk your\_project\_name.apk

## Some of App markets

* [](https://play.google.com/store)

#### Google play

* [](http://www.phoload.com/)

#### phoload

* [http://androidmarkets.com/wp-content/uploads/2012/02/logotipo-e1424956156930.png](https://m.aptoide.com/installer)

#### APTOiDE

* [](https://www.amazon.com/mobile-apps/b?node=2350149011)

#### Amazon AppStore

* [](https://www.1mobile.com/)

#### 1mobile

* [](http://www.insydemarket.com/)

#### Insyde Market

* [](https://store.yandex.com/)

#### Yandex store

#### F-Droid

* [](https://www.samsung.com/in/apps/mobile/galaxyapps/?redirectionId=http://apps.samsung.com/mercury/main/getMain.as)

#### Samsung Galaxy AppStore