Open Handset Alliance (OHA)

It is a consortium of 84 firms to develop open standards for mobile devices. Member firms include HTC, Sony, Dell, Intel, Motorola, Qualcomm, Texas Instruments, Google, Samsung Electronics, LG Electronics, T-Mobile, Sprint Corporation, Nvidia, and Wind River Systems.

The OHA was established on 5 November 2007, led by Google with 34 members, including mobile handset makers, application developers, some mobile carriers and chip makers. Android, the software of the alliance (first developed by Google in 2007), is based on an open-source license and has competed against mobile platforms from Apple, Microsoft, Nokia (Symbian), HP (formerly Palm), Samsung Electronics / Intel, and BlackBerry.

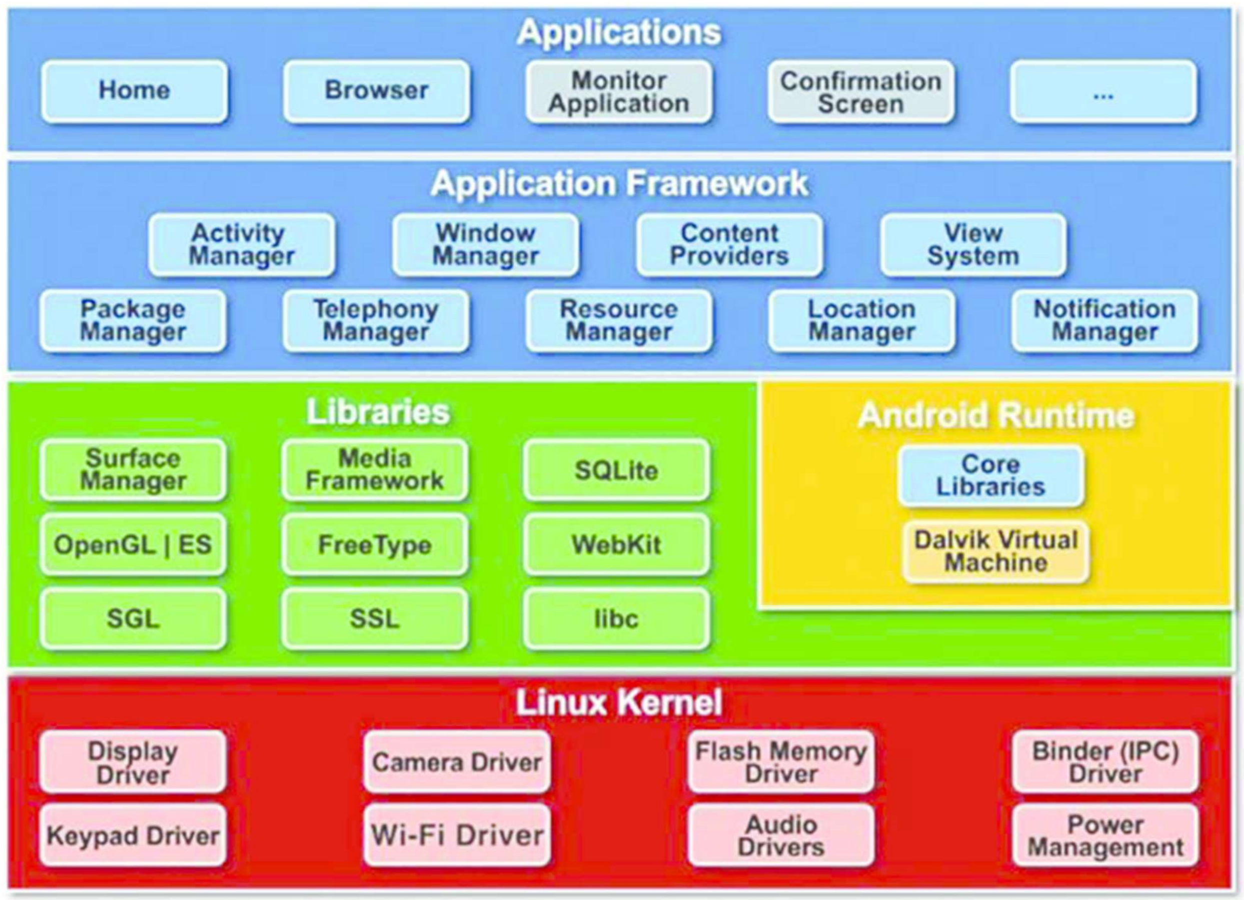
As part of its efforts to promote a unified Android platform, OHA members are contractually forbidden from producing devices that are based on incompatible forks of Android.

Android Platform

The Android platform is a platform for mobile devices that uses a modified Linux kernel. The Android Platform was introduced by the Open Handset Alliance in November of 2007. Most applications that run on the Android platform are written in the Java programming language.

The Android Platform was launched in 2007 by the Open Handset Alliance, an alliance of prominent companies that includes Google, HTC, Motorola, Texas Instruments and others. Although most of the applications that run on the Android Platform are written in Java, there is no Java Virtual Machine. Instead, the Java classes are first compiled into what are known as Dalvik Executables and run on the Dalvik Virtual Machine.

Android is an open development platform. However, it is not open in the sense that everyone can contribute while a version is under development. This is all done behind closed-doors at Google. Rather, the openness of Android starts when its source code is released to the public after it is finalized. This means once it is released anyone interested can take the code and alter it as they see fit.



libraries

1. *Libc*: it is c standard lib.
2. *SSL*: Secure Socket Layer for security
3. *SGL*: 2D picture engine where SGL is “Scalable Graphics Library”
4. *OpenGL|ES*: 3D image engine
5. *Media Framework*: essential part of Android multi-media
6. *SQLite*: Embedded database
7. *Web Kit*: Kernel of web browser
8. *Free Type*: Bitmap and Vector
9. *Surface Manager*: Manage different windows for different applications

To create an application for the platform, a developer requires the Android SDK, which includes tools and APIs. To shorten development time, Android developers typically integrate the SDK into graphical user IDEs (Integrated Development Environments). Beginners can also make use of the App Inventor, an application for creating Android apps that can be accessed online.

Android SDK

A software development kit that enables developers to create applications for the Android platform. The Android SDK includes sample projects with source code, development tools, an emulator, and required libraries to build Android applications. Applications are written using the Java programming language and run on Dalvik, a custom virtual machine designed for embedded use which runs on top of a Linux kernel.

The Android software development kit (SDK) includes a comprehensive set of development tools. These include a debugger, libraries, a handset emulator ,documentation, sample code, and tutorials. Currently supported development platforms include computers running Linux (any modern desktop Linux distribution), Mac OS X 10.5.8 or later, and Windows 7 or later. As of March 2015, the SDK is not available on Android itself, but software development is possible by using specialized Android applications.

Software Development Kit (SDK) is a collection of Software Development tools in one installable package. This SDK is also used with *Android* which helps to download the tools, the latest versions of Android

Enhancements to Android's SDK go hand-in-hand with the overall Android platform development. The SDK also supports older versions of the Android platform in case developers wish to target their applications at older devices. Development tools are downloadable components, so after one has downloaded the latest version and platform, older platforms and tools can also be downloaded for compatibility testing.

Android applications are packaged in .apk format and stored under /data/app folder on the Android OS (the folder is accessible only to the root user for security reasons). APK package contains .dex files (compiled byte code files called Dalvik executables), resource files, etc.

What is the Android SDK?

Every time Google releases a new version, the corresponding SDK is also released. In order to work with Android, the developers must download and install each version's SDK for the particular device.

The Android SDK (Software Development Kit) is a set of development tools that are used to develop applications for the Android platform.

This SDK provides a selection of tools that are required to build Android applications and ensures the process goes as smoothly as possible. Whether you create an application using *Java*, you need the SDK to get it to run on any Android device. You can also use an emulator in order to test the applications that you have built.

Nowadays, the Android SDK also comes bundled with Android Studio, the integrated development environment where the work gets done and many of the tools are now best accessed or managed.

Note: You can download the Android SDK independently.

Anatomy of an Android

Android Studio Projects

You're probably using Android Studio to organize the files and code that make up your app, or maybe you're using another IDE like Eclipse. You can build everything manually using basic text editors and the command line as well, but because of how complicated app projects are, most people use an IDE.

In any case, your Android app is split up into several directories, and you'll use your IDE to access all of them.

AndroidManifest.xml

The `AndroidManifest.xml` file contains a bunch of properties that you'll need to set when you eventually deploy your app to the Play Store or on other phones. This is where stuff like the name of your app and its permissions gets set.

We'll come back to this file when we talk about deploying your app.

Source Code

Android apps are written in Java (technically it's not quite Java, but let's not worry too much about that), so the source code of Android apps is stored in `.java` files, just like you're already used to.

The `.java` files are stored in whatever package you chose when you created your app project. The entry point of an Android app is the main activity class: by default it's `MainActivity.java`. We'll talk more about activities in a minute, but for now just know that your code will go in a bunch of `.java` files inside your project.

Resources

The `res` directory contains non-code files that are needed to run your app. Stuff like images and property files go here.

There are a few subdirectories under the `res` folder:

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	**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.
7	**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.

We'll talk more about resources as we need them in the other tutorials, but for now just know that non-code stuff goes here.

Android Concepts

Now that we know how our project is laid out, let's talk about the structure of an Android app.

Views

Views are things like buttons, text fields, and labels. They're individual components that the user can view and interact with. Views are the basic building blocks that make up your app. You can think of a view as a widget, or a component, or an element, depending on which UI library you've used before.

Android views are represented by classes. For example, a button is represented by the `Button` class. To create a button, you'd create an instance of the `Button` class (import android.widget.Button).

Our hello world app uses a `TextView` to show a label and a `Button` view to show a button.

Layouts

Views are put together into layouts, which decide how the views are shown on screen. A layout decides the placement and size of the views it holds. You can think of a layout as a single screen in your app.

In Anrdoid, layouts are containers that hold views (as opposed to being a property on a container, like in Swing or CSS). For example, the `LinearLayout` class represents a layout that dispays views in a single vertical column or horizontal row.

Our example uses a `LinearLayout` to position the views in our app.

Activities

The code that runs an Android app is called an **activity**. An app can be divided into several activities, and there's usually one activity per screen. An activity is also the entry point (think `main` method) of an app.

Activities have a **lifecycle**, which is a series of events that happen to an activity: stuff like creation, pausing, resuming, and exiting.

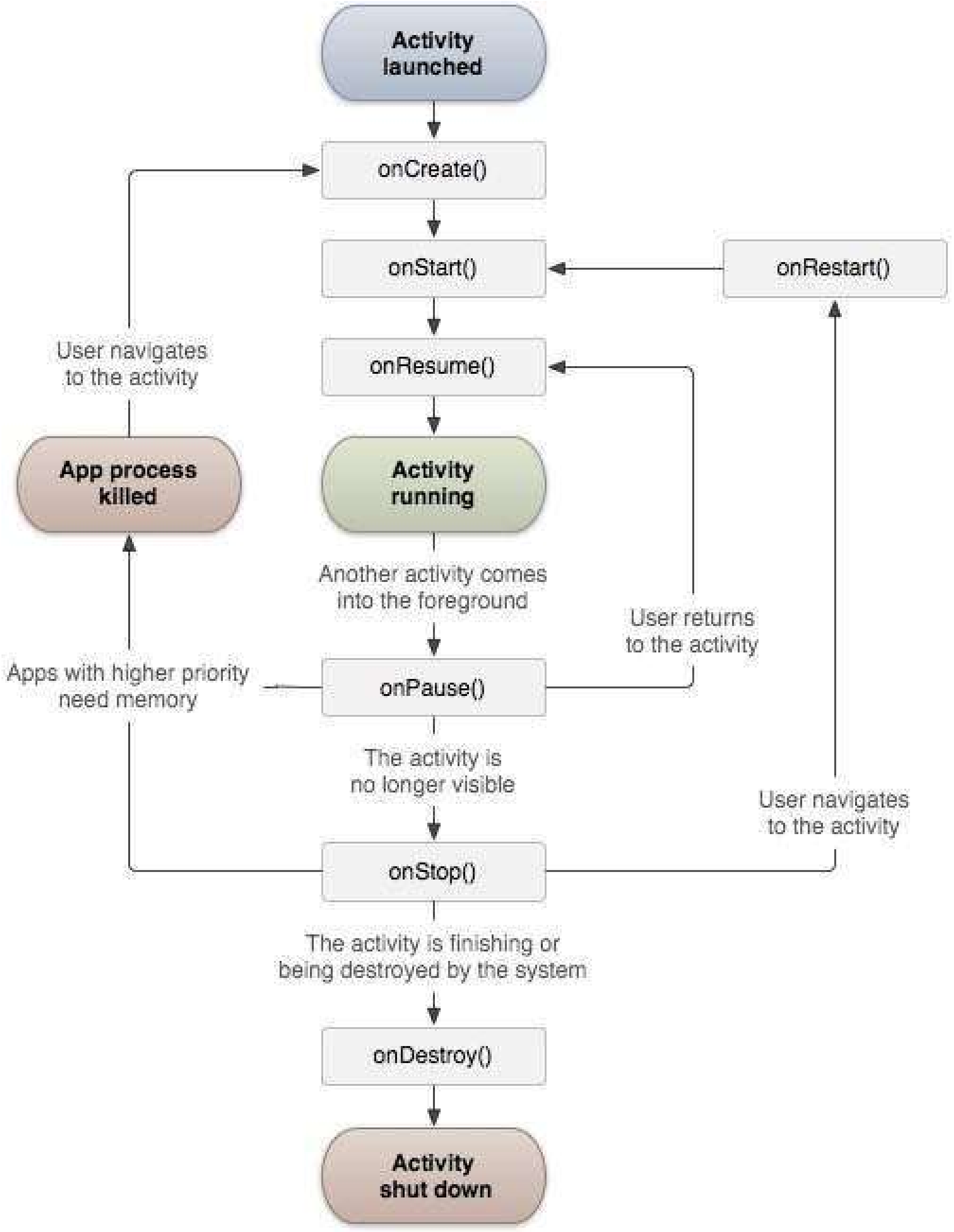
Activities are represented by the `Activity` class.

An activity usually loads a layout in its `onCreate()` function and sets up stuff like event listeners and views. For example, our hello world app defines a `MainActivity` class that overrides the `onCreate()` function, which is called when the app is first run. That code loads our layout and sets up a click listener on the button.

The `AndroidManifest.xml` file tells Android which activity to run when your app is opened.

Android Activities

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)


Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

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:

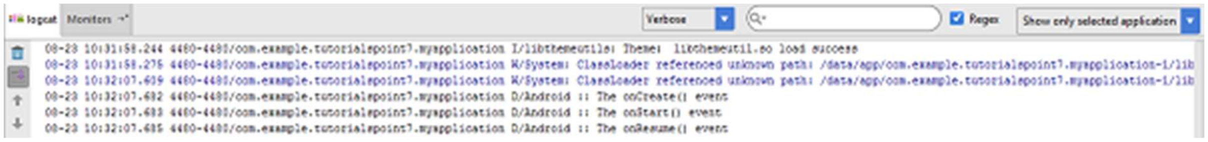
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.tutorialspoint7.myapplication">

    <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 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 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
```



```
08-23 10:31:58.244 4480-4480/com.example.tutorialspoint7.myapplication I/libthemeutils: Theme:  libthemeutil.so load success
08-23 10:31:58.276 4480-4480/com.example.tutorialspoint7.myapplication W/System: ClassLoader referenced unknown path: /data/app/com.example.tutorialspoint7.myapplication-1/lib
08-23 10:32:07.609 4480-4480/com.example.tutorialspoint7.myapplication W/System: ClassLoader referenced unknown path: /data/app/com.example.tutorialspoint7.myapplication-1/lib
08-23 10:32:07.682 4480-4480/com.example.tutorialspoint7.myapplication D/Android :: The onCreate() event
08-23 10:32:07.683 4480-4480/com.example.tutorialspoint7.myapplication D/Android :: The onStart() event
08-23 10:32:07.685 4480-4480/com.example.tutorialspoint7.myapplication 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 on 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 Intents

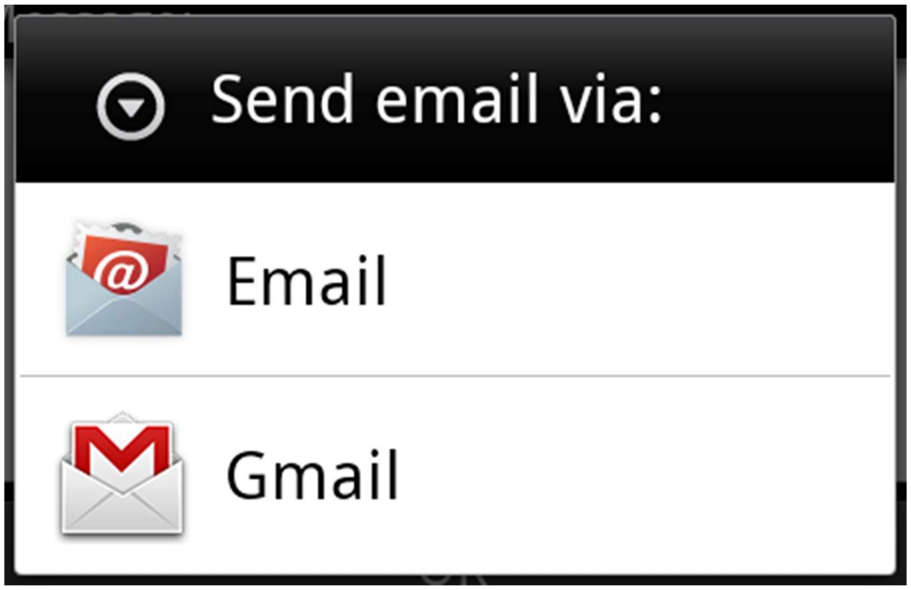
An Android **Intent** is an abstract description of an operation to be performed. It can be used with **startActivity** to launch an Activity, **broadcastIntent** to send it to any interested BroadcastReceiver components, and **startService(Intent)** or **bindService(Intent, ServiceConnection, int)**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

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.

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

Extras

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

Flags

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

Sr.No	Flags & Description
1	**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.

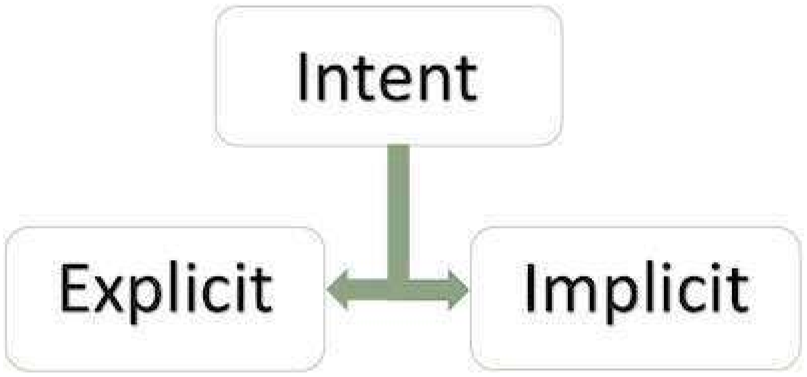
Component Name

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().

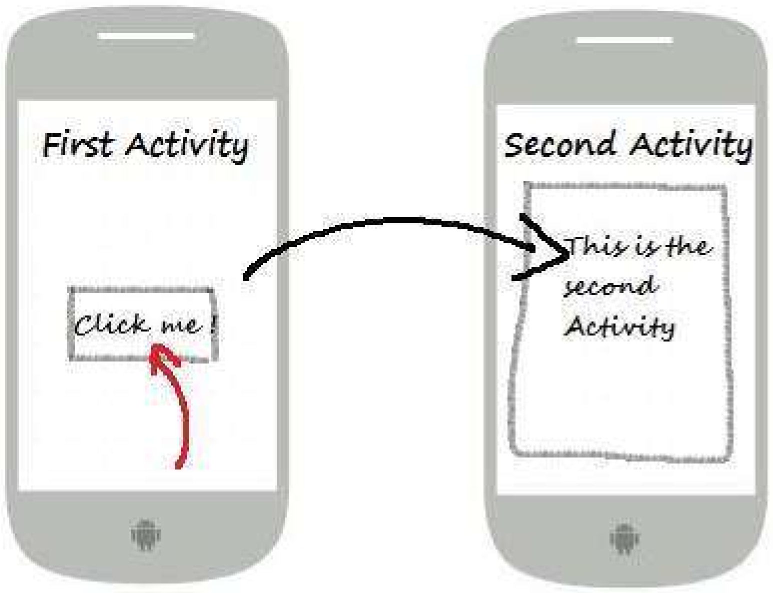
Types of Intents

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 −

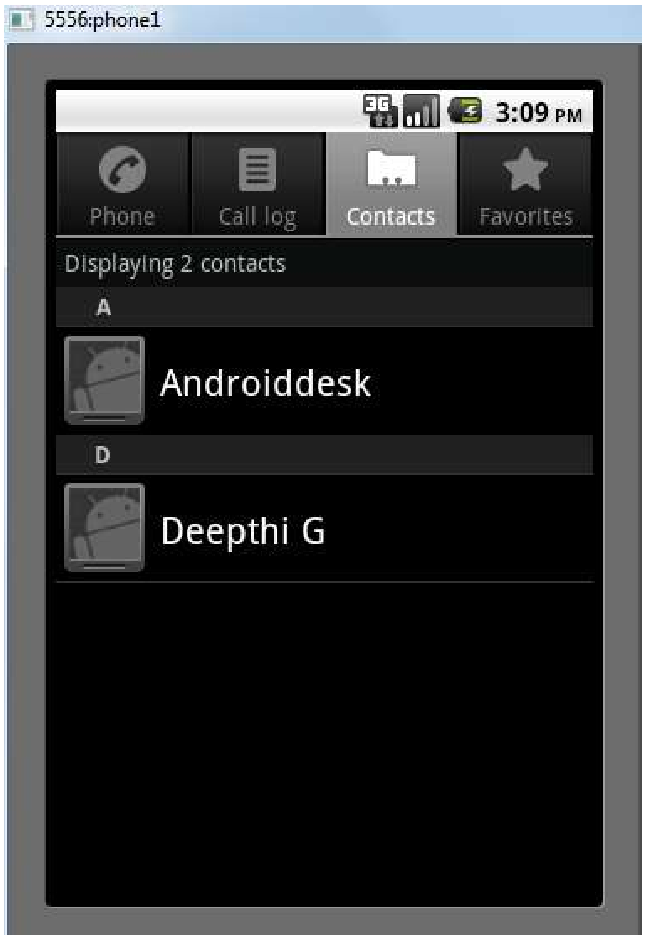
```
// Explicit Intent by specifying its class name
Intent i = new Intent(FirstActivity.this, SecondActivity.class);

// 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.kscpac.org"));
            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:9979271572"));
            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>
```

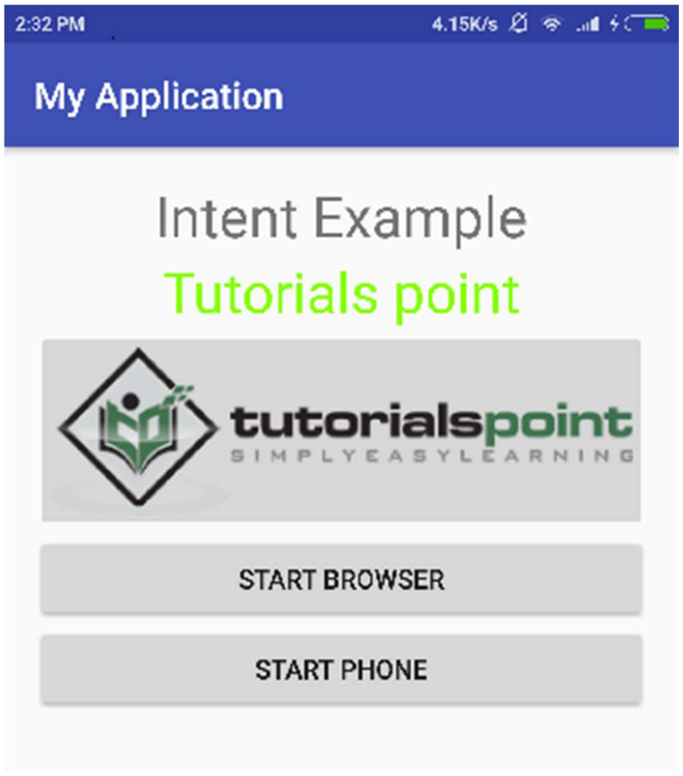
Following will be the content of **res/values/strings.xml** to define two new constants −

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="app_name">My Applicaiton</string>
</resources>
```

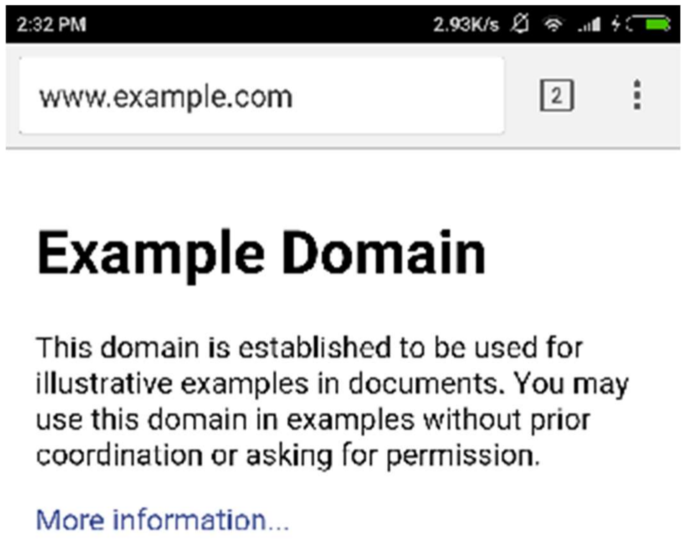
Following is the default content of **AndroidManifest.xml** −

```
<?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 −



Example Domain

This domain is established to be used for illustrative examples in documents. You may use this domain in examples without prior coordination or asking for permission.

More information...

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 −

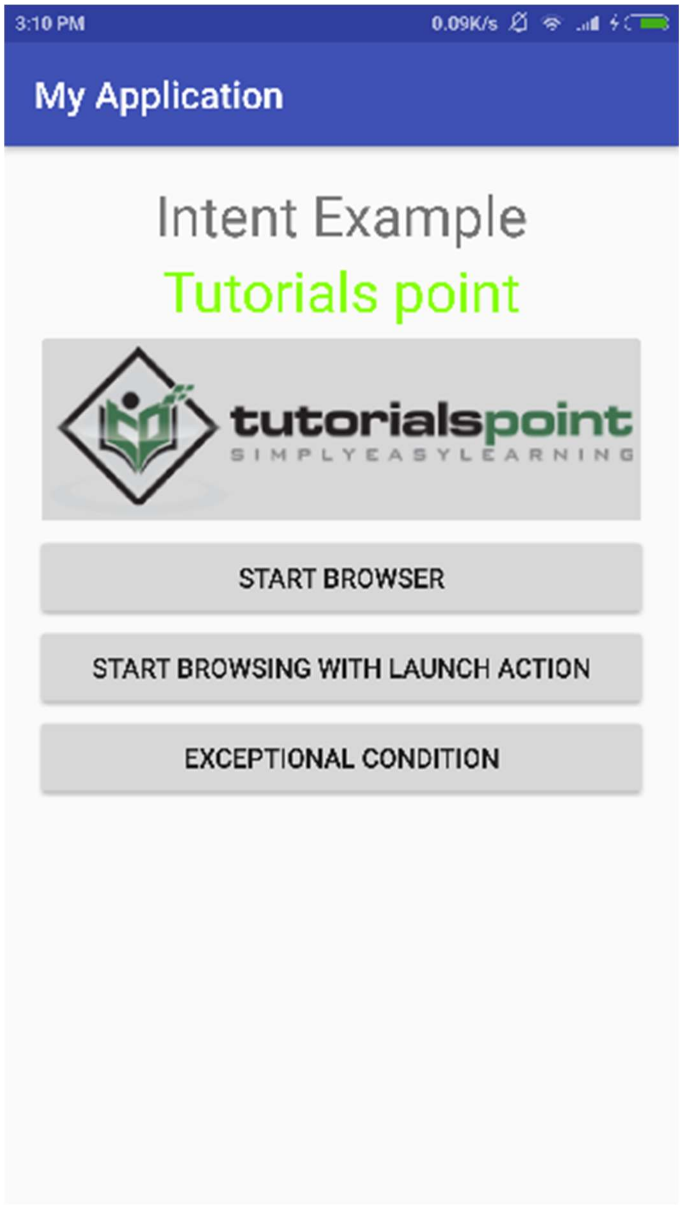
```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="app_name">My Application</string>
</resources>
```

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

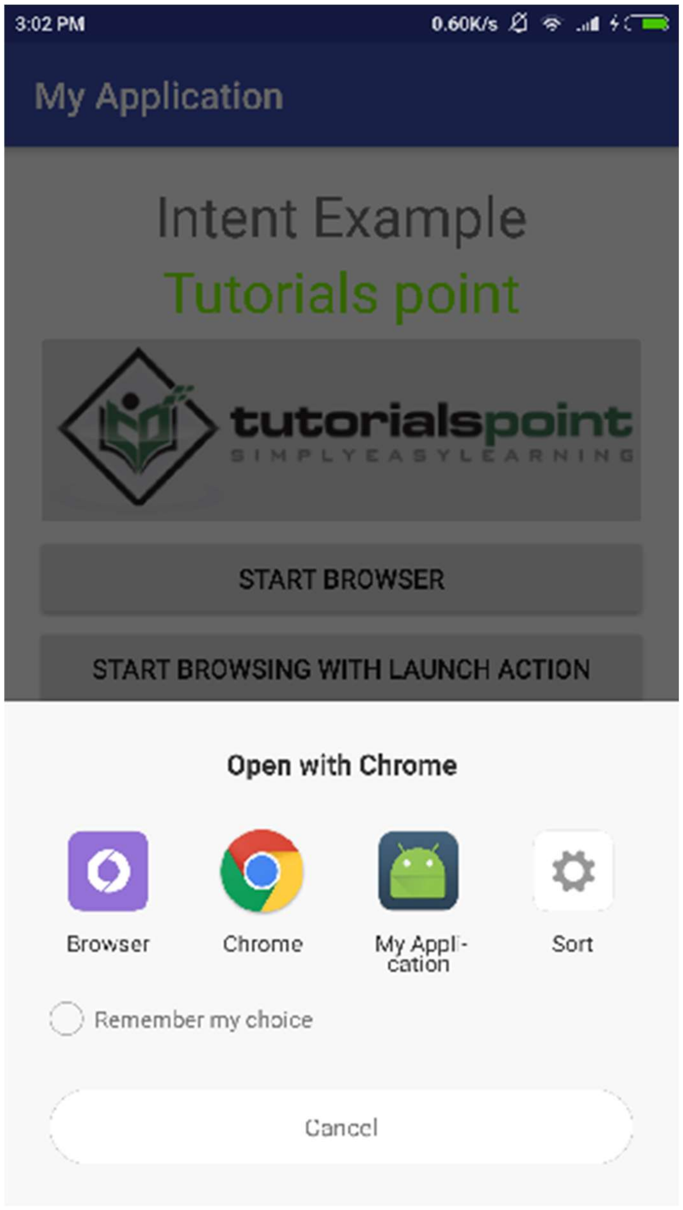
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.example.tutorialspoint7.myapplication">

   <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 ▶icon from the toolbar. Android Studio installs the app on your

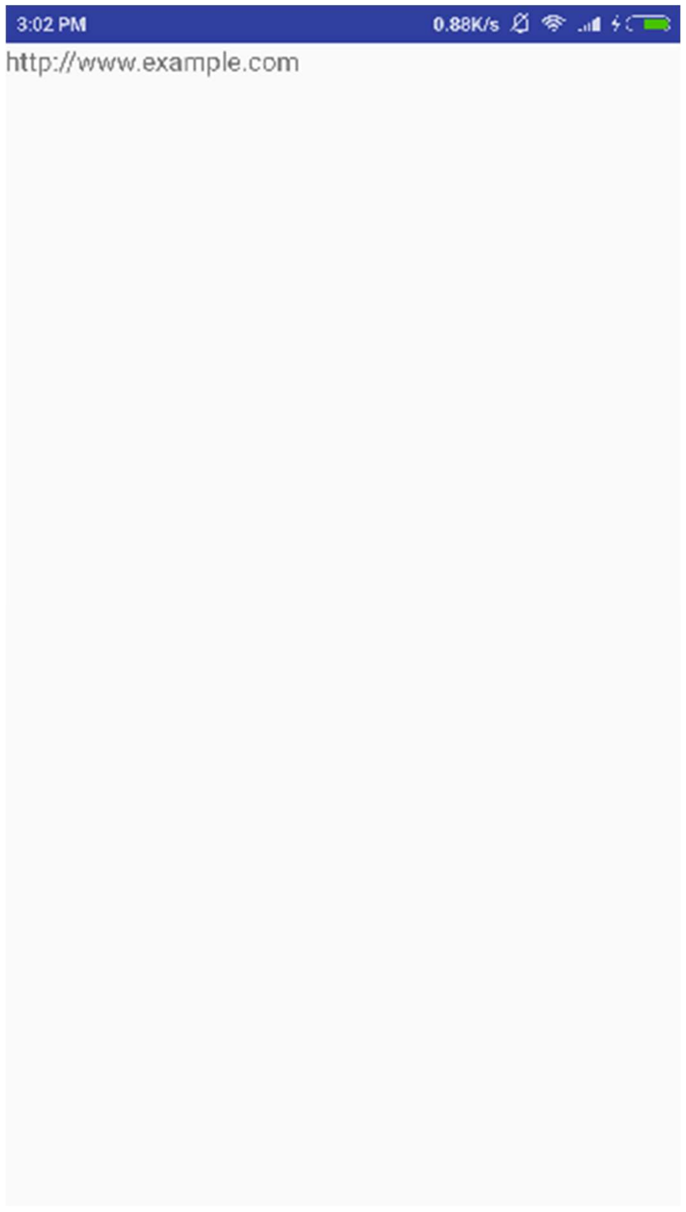
AVD and starts it and if everything is fine with your setup and application, it will display following Emulator window −



Now 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.



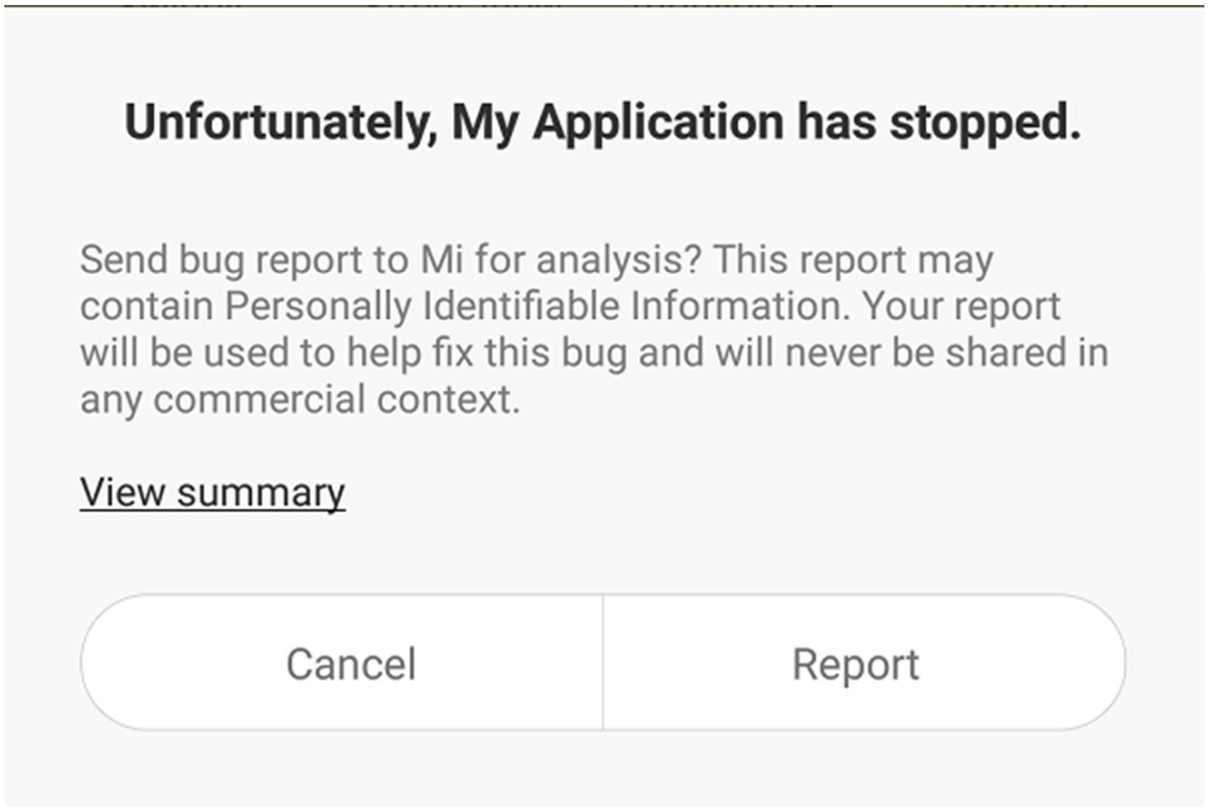
Now if you select Browser, then Android will launch web browser and open example.com website but if you select IndentDemo option then Android will launch CustomActivity which does nothing but just capture passed data and displays in a text view as follows −



http://www.example.com

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 −


Unfortunately, My Application has stopped.

Send bug report to Mi for analysis? This report may contain Personally Identifiable Information. Your report will be used to help fix this bug and will never be shared in any commercial context.

Broadcast Intents

Another type of Intent, the Broadcast Intent, is a system wide intent that is sent out to all applications that have registered an “interested” Broadcast Receiver. The Android system, for example, will typically send out Broadcast Intents to indicate changes in device status such as the completion of system start up, connection of an external power source to the device or the screen being turned on or off.

A Broadcast Intent can be normal (asynchronous) in that it is sent to all interested Broadcast Receivers at more or less the same time, or ordered in that it is sent to one receiver at a time where it can be processed and then either aborted or allowed to be passed to the next Broadcast Receiver.

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.

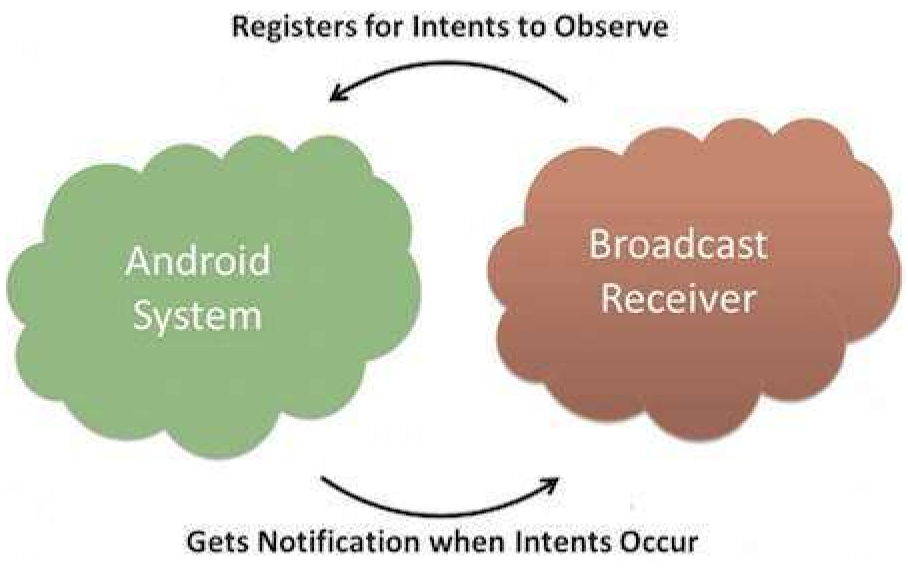
Creating the Broadcast Receiver

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.


Registers for Intents to Observe
Android System
Broadcast Receiver
Gets Notification when Intents Occur

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();
   }
}
```

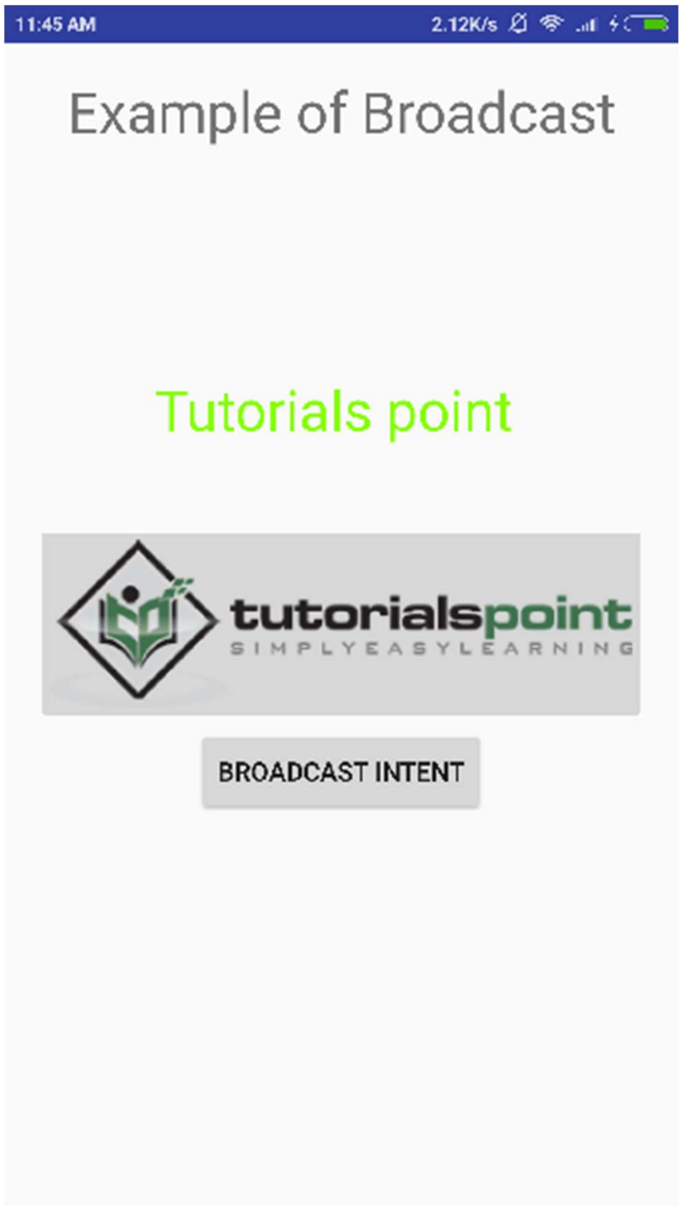
Following will the modified content of *AndroidManifest.xml* file. Here we have added <receiver.../> tag to include our service:

```
<?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>
```

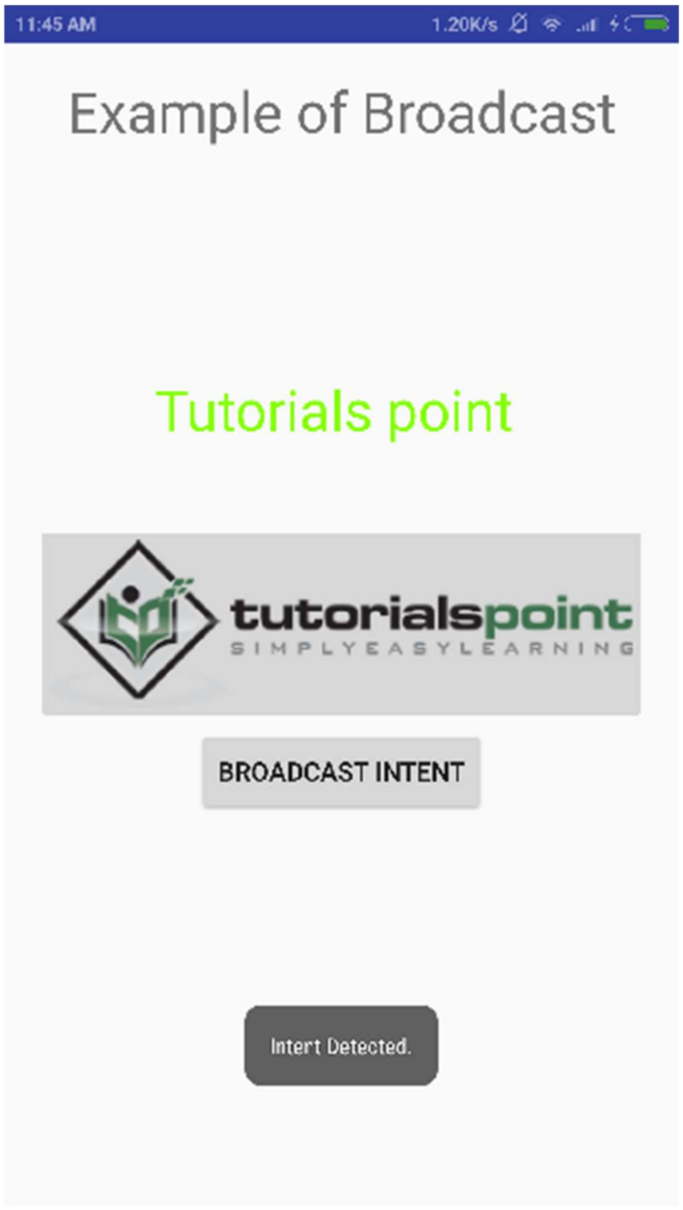
Following will be the content of **res/layout/activity_main.xml** file to include a button to broadcast our custom intent −

```
<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 ▶icon 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.

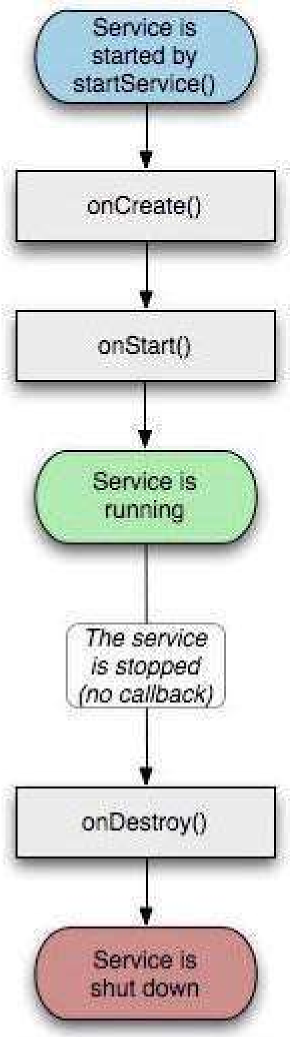
Android Services

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 –

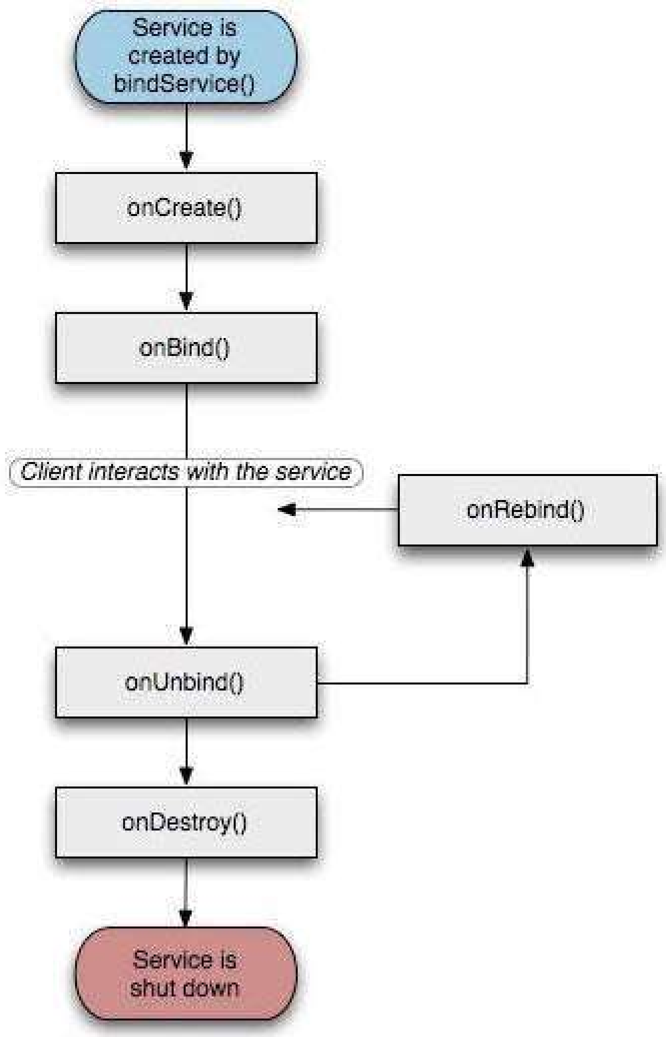
Sr.No.	State & Description
1	**Started**

	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)*


Service is started by startService()
onCreate()
onStart()
Service is running
The service is stopped (no callback)
onDestroy()
Service is shut down

UN Bounded Service



Bounded services

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();
   }
}
```

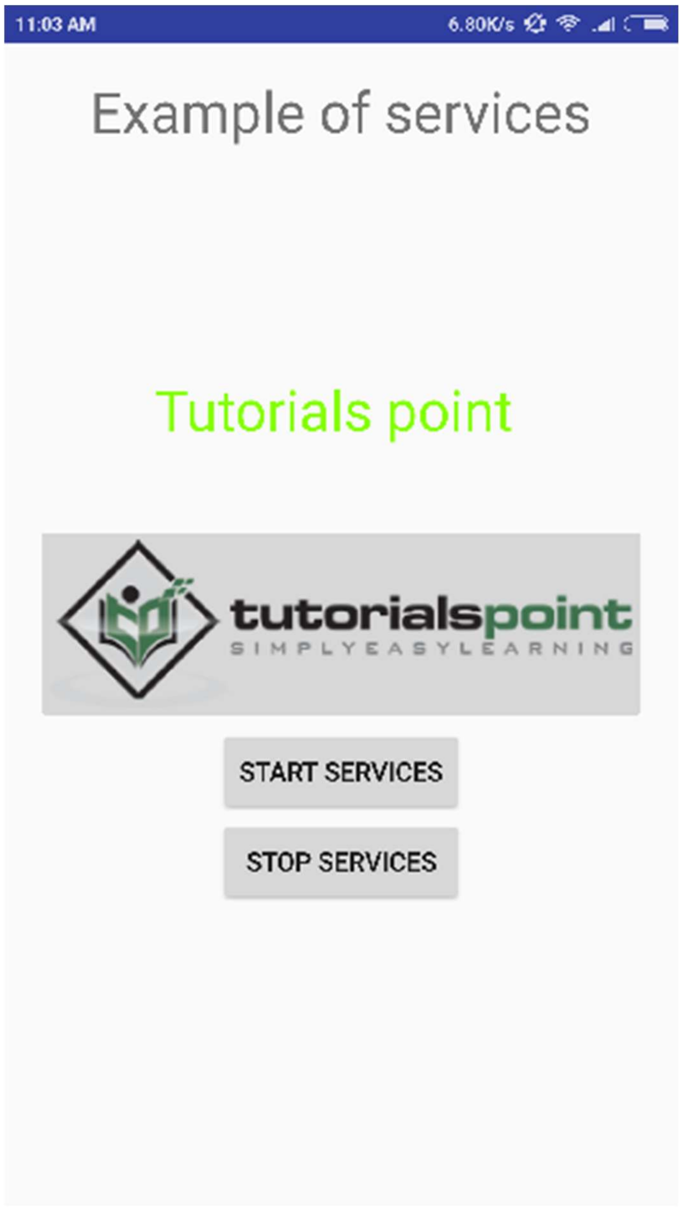
Following will the modified content of *AndroidManifest.xml* file. Here we have added <service.../> tag to include our service −

```
<?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>
```

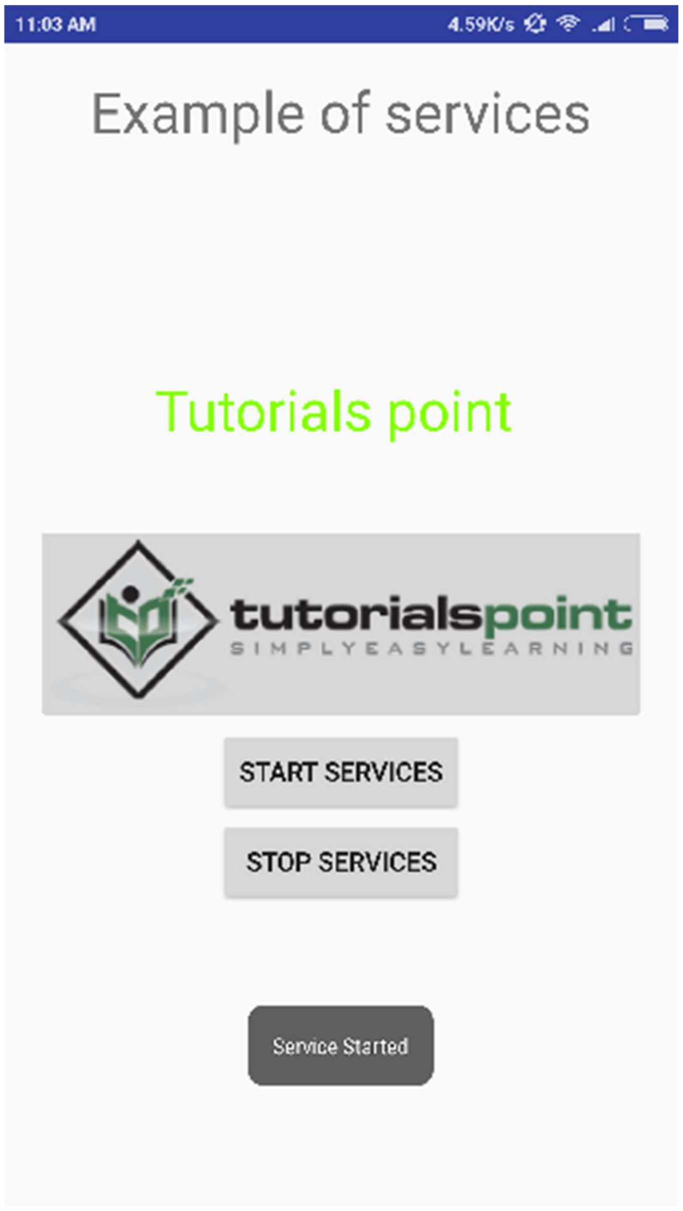
Following will be the content of **res/layout/activity_main.xml** file to include two buttons −

```
<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 ▶icon 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 −



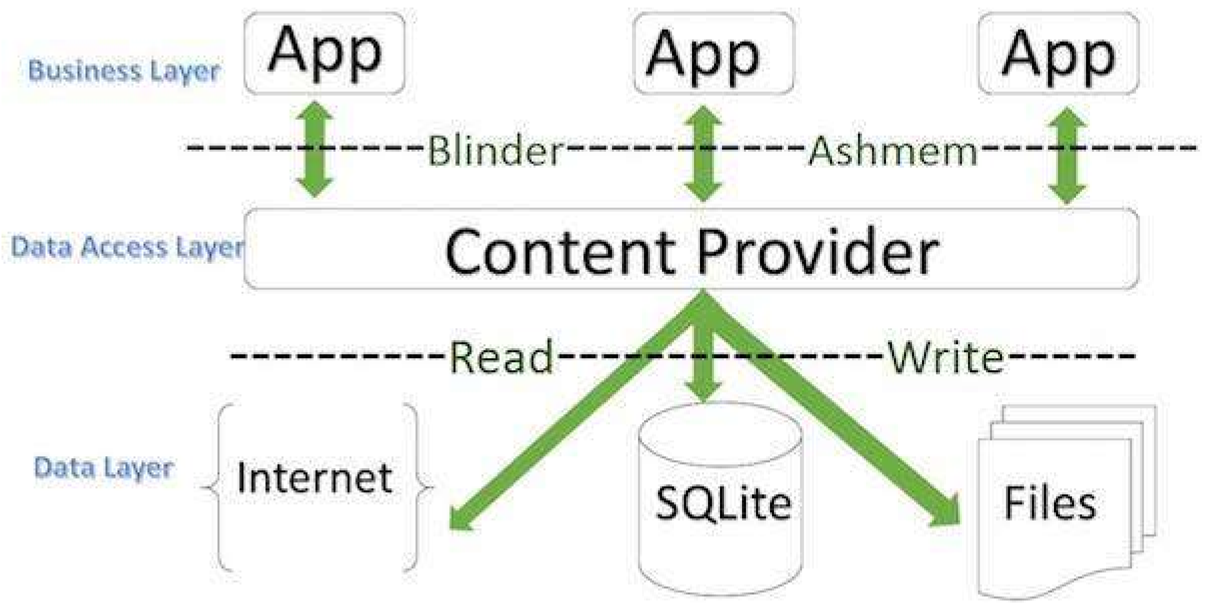
Now to start your service, let's click on **Start Service** button, this will start the service and as per our programming in *onStartCommand()* method, a message *Service Started* will appear on the bottom of the the simulator as follows −



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

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://

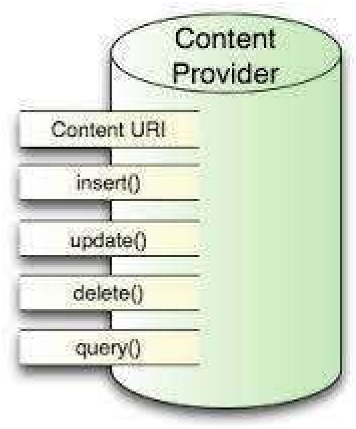
2	**Authority** This specifies the name of the content provider, for example *contacts*, *browser* etc. For third-party content providers, this could be the fully qualified name, such as *com.tutorialspoint.statusprovider*
3	**data_type** This indicates the type of data that this particular provider 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>
```

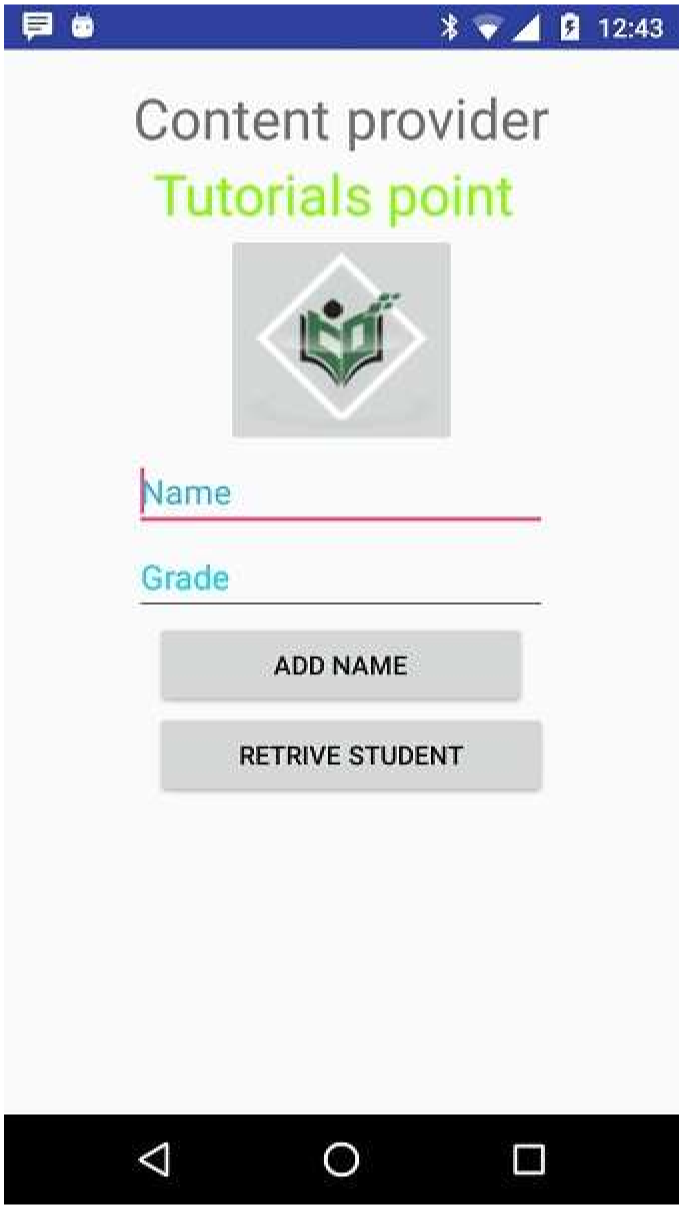
Following will be the content of **res/layout/activity_main.xml** file−

```
<?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>
```

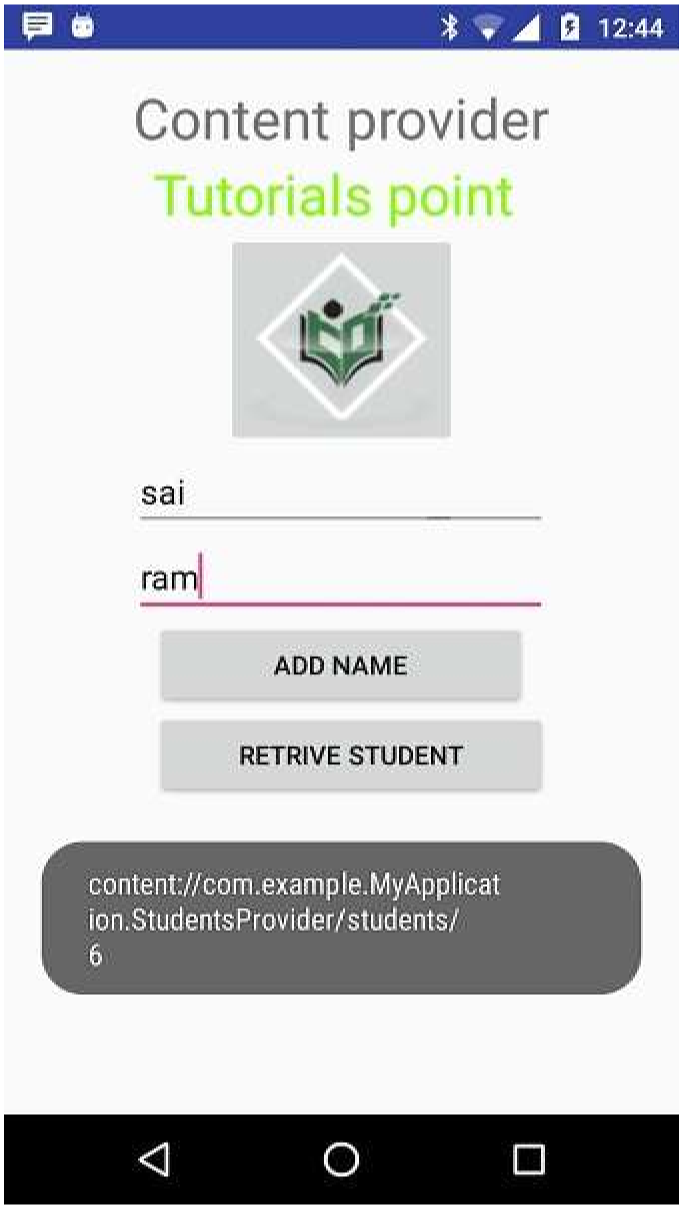
Make sure you have following content of **res/values/strings.xml** file:

```
<?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 ▶icon 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.



content://com.example.MyApplication.StudentsProvider/students/6

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.

The Application Manifest

The **AndroidManifest.xml file** *contains information of your package*, including components of the application such as activities, services, broadcast receivers, content providers etc.

It performs some other tasks also:

- It is **responsible to protect the application** to access any protected parts by providing the permissions.
- It also **declares the android api** that the application is going to use. This is the required xml file for all the android application and located inside the root directory.

A simple AndroidManifest.xml file looks like this:

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

  <uses-sdk
    android:minSdkVersion="8"
    android:targetSdkVersion="15" />

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme" >
    <activity
      android:name=".MainActivity"
      android:label="@string/title_activity_main" >
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>

</manifest>
```

Elements of the AndroidManifest.xml file

The elements used in the above xml file are described below.

<manifest>

manifest is the root element of the AndroidManifest.xml file. It has **package** attribute that describes the package name of the activity class.

<application>

application is the subelement of the manifest. It includes the namespace declaration. This element contains several subelements that declares the application component such as activity etc.

The commonly used attributes are of this element are **icon**, **label**, **theme** etc.

android:icon represents the icon for all the android application components.

android:label works as the default label for all the application components.

android:theme represents a common theme for all the android activities.

<activity>

activity is the subelement of application and represents an activity that must be defined in the AndroidManifest.xml file. It has many attributes such as label, name, theme, launchMode etc.

android:label represents a label i.e. displayed on the screen.

android:name represents a name for the activity class. It is required attribute.

<intent-filter>

intent-filter is the sub-element of activity that describes the type of intent to which activity, service or broadcast receiver can respond to.

<action>

It adds an action for the intent-filter. The intent-filter must have at least one action element.

<category>

It adds a category name to an intent-filter.

Application Context

When an application is compiled, a class named R is created that contains references to the application resources. The application manifest file and these resources combine to create what is known as the Application Context. This context, represented by the Android Context class, may be used in the application code to gain access to the application resources at runtime. In addition, a wide range of methods may be called on an application's context to gather information and make changes to the application's environment at runtime.

Terminologies Related to Android

- **XML**

In Android, XML is used for designing the application's UI like creating layouts, views, buttons, text fields etc. and also used in parsing data feeds from the internet.

- **View**

A view is an UI which occupies rectangular area on the screen to draw and handle user events.

- **Layout**

Layout is the parent of view. It arranges all the views in a proper manner on the screen.

- **Activity**

An activity can be referred as your device's screen which you see. User can place UI elements in any order in the created window of user's choice.

- **Emulator**

An emulator is an Android virtual device through which you can select the target Android0020version or platform to run and test your developed application.

- **Manifest file**

Manifest file acts as a metadata for every application. This file contains all the essential information about the application like app icon, app name, launcher activity, and required permissions etc.

- **Service**

Service is an application component that can be used for long-running background processes. It is not bounded with any activity as there is no UI. Any other application component can start a service and this service will continue to run even when the user switches from one application to another.

- **Broadcast Receiver**

Broadcast Receiver is another building block of Android application development which allows you to register for system and application events. It works in such a way that, when the event triggers for the first time all the registered receivers through this broadcast receiver will get notified for all the events by Android Runtime.

- **Content Providers**

Content Providers are used to share data between two applications. This can be implemented in two ways:

1. When you want to implement the existing content provider in another application.
2. When you want to create a new content provider that can share its data with other applications

- **Intent**

Intent is a messaging object which can be used to communicate between two or more components like activities, services, broadcast receiver etc. Intent can also be used to start an activity or service or to deliver a broadcast messages.

Android Manifest File and its common settings

Every app project must have an `AndroidManifest.xml` file (with precisely that name) at the root of the project source set. The manifest file describes essential information about your app to the Android build tools, the Android operating system, and Google Play.

Among many other things, the manifest file is required to declare the following:

- The app's package name, which usually matches your code's namespace. The Android build tools use this to determine the location of code entities when building your project. When packaging the app, the build tools replace this value with the application ID from the Gradle build files, which is used as the unique app identifier on the system and on Google Play. Read more about the package name and app ID.
- The components of the app, which include all activities, services, broadcast receivers, and content providers. Each component must define basic properties such as the name of its Kotlin or Java class. It can also declare capabilities such as which device configurations it can handle, and intent filters that describe how the component can be started. Read more about app components.
- The permissions that the app needs in order to access protected parts of the system or other apps. It also declares any permissions that other apps must have if they want to access content from this app. Read more about permissions.
- The hardware and software features the app requires, which affects which devices can install the app from Google Play. Read more about device compatibility.

If you're using Android Studio to build your app, the manifest file is created for you, and most of the essential manifest elements are added as you build your app (especially when using code templates).

File features

The following sections describe how some of the most important characteristics of your app are reflected in the manifest file.

Package name and application ID

The manifest file's root element requires an attribute for your app's package name (usually matching your project directory structure—the Java namespace).

For example, the following snippet shows the root `<manifest>` element with the package name `"com.example.myapp"`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapp"
    android:versionCode="1"
    android:versionName="1.0" >
    ...
</manifest>
```

App components

For each app component that you create in your app, you must declare a corresponding XML element in the manifest file:

- `<activity>` for each subclass of `Activity`.
- `<service>` for each subclass of `Service`.
- `<receiver>` for each subclass of `BroadcastReceiver`.
- `<provider>` for each subclass of `ContentProvider`.

If you subclass any of these components without declaring it in the manifest file, the system cannot start it.

The name of your subclass must be specified with the `name` attribute, using the full package designation. For example, an `Activity` subclass can be declared as follows:

```
<manifest ... >
    <application ... >
        <activity android:name="com.example.myapp.MainActivity" ... >
        </activity>
    </application>
</manifest>
```

However, if the first character in the `name` value is a period, the app's package name (from the `<manifest>` element's `package` attribute) is prefixed to the name. For example, the following activity name is resolved to `"com.example.myapp.MainActivity"`:

```
<manifest package="com.example.myapp" ... >
    <application ... >
        <activity android:name=".MainActivity" ... >
            ...
        </activity>
    </application>
</manifest>
```

If you have app components that reside in sub-packages (such as in `com.example.myapp.purchases`), the `name` value must add the missing sub-package names (such as `".purchases.PayActivity"`) or use the fully-qualified package name.

Intent filters

App activities, services, and broadcast receivers are activated by *intents*. An intent is a message defined by an Intent object that describes an action to perform, including the data to be acted upon, the category of component that should perform the action, and other instructions.

When an app issues an intent to the system, the system locates an app component that can handle the intent based on *intent filter* declarations in each app's manifest file. The system launches an instance of the matching component and passes the Intent object to that component. If more than one app can handle the intent, then the user can select which app to use.

An app component can have any number of intent filters (defined with the <intent-filter> element), each one describing a different capability of that component.

Permissions

Android apps must request permission to access sensitive user data (such as contacts and SMS) or certain system features (such as the camera and internet access). Each permission is identified by a unique label. For example, an app that needs to send SMS messages must have the following line in the manifest:

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

Beginning with Android 6.0 (API level 23), the user can approve or reject some app permisions at runtime. But no matter which Android version your app supports, you must declare all permission requests with a `<uses-permission>` element in the manifest. If the permission is granted, the app is able to use the protected features. If not, its attempts to access those features fail.

Your app can also protect its own components with permissions. It can use any of the permissions that are defined by Android, as listed in `android.Manifest.permission`, or a permission that's declared in another app. Your app can also define its own permissions. A new permission is declared with the `<permission>` element.

<uses-sdk>

Each successive platform version often adds new APIs not available in the previous version. To indicate the minimum version with which your app is compatible, your manifest must include the `<uses-sdk>` tag and its `minSdkVersion` attribute.

However, beware that attributes in the `<uses-sdk>` element are overridden by corresponding properties in the `build.gradle` file. So if you're using Android Studio, you must specify the `minSdkVersion` and `targetSdkVersion` values there instead:

```
android {
  defaultConfig {
    applicationId 'com.example.myapp'

    // Defines the minimum API level required to run the app.
    minSdkVersion 15

    // Specifies the API level used to test the app.
    targetSdkVersion 26

    ...
  }
}
```

Manifest elements reference

The following table provides links to reference documents for all valid elements in the `AndroidManifest.xml` file.

<action>	Adds an action to an intent filter. The intent-filter must have at least one action element.
<activity>	Declares an activity component. **activity** is the subelement of application and represents an activity that must be defined in the AndroidManifest.xml file. It has many attributes such as label, name, theme, launchMode etc. **android:label** represents a label i.e. displayed on the screen. **android:name** represents a name for the activity class. It is required attribute.
<activity-alias>	Declares an alias for an activity.
<application>	The declaration of the application. **application** is the subelement of the manifest. It includes the namespace declaration. This element contains several subelements that declares the application component such as activity etc. The commonly used attributes are of this element are **icon**, **label**, **theme** etc. **android:icon** represents the icon for all the android application components. **android:label** works as the default label for all the application components.

	android:theme represents a common theme for all the android activities.
<category>	Adds a category name to an intent filter.
<compatible-screens>	Specifies each screen configuration with which the application is compatible.
<data>	Adds a data specification to an intent filter.
<grant-uri-permission>	Specifies the subsets of app data that the parent content provider has permission to access.
<instrumentation>	Declares an `Instrumentation` class that enables you to monitor an application's interaction with the system.
<intent-filter>	Specifies the types of intents that an activity, service, or broadcast receiver can respond to.
<manifest>	The root element of the AndroidManifest.xml file. It has **package** attribute that describes the package name of the activity class.
<meta-data>	A name-value pair for an item of additional, arbitrary data that can be supplied to the parent component.
<path-permission>	Defines the path and required permissions for a specific subset of data within a content provider.
<permission>	Declares a security permission that can be used to limit access to specific components or features of this or other applications.
<permission-group>	Declares a name for a logical grouping of related permissions.
<permission-tree>	Declares the base name for a tree of permissions.
<provider>	Declares a content provider component.
<receiver>	Declares a broadcast receiver component.
<service>	Declares a service component.
<supports-gl-texture>	Declares a single GL texture compression format that the app supports.

<supports-screens>	Declares the screen sizes your app supports and enables screen compatibility mode for screens larger than what your app supports.
<uses-configuration>	Indicates specific input features the application requires.
<uses-feature>	Declares a single hardware or software feature that is used by the application.
<uses-library>	Specifies a shared library that the application must be linked against.
<uses-permission>	Specifies a system permission that the user must grant in order for the app to operate correctly.
<uses-permission-sdk-23>	Specifies that an app wants a particular permission, but only if the app is installed on a device running Android 6.0 (API level 23) or higher.
<uses-sdk>	Lets you express an application's compatibility with one or more versions of the Android platform, by means of an API level integer.

Example manifest file

The XML below is a simple example `AndroidManifest.xml` that declares two activities for the app.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="1"
    android:versionName="1.0"
    package="com.example.myapp">

    <!-- Beware that these values are overridden by the build.gradle file -->
    <uses-sdk android:minSdkVersion="15" android:targetSdkVersion="26" />

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">

        <!-- This name is resolved to com.example.myapp.MainActivity
             based upon the package attribute -->
        <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=".DisplayMessageActivity"
            android:parentActivityName=".MainActivity" />
    </application>
</manifest>
```

Managing Application resources in a hierarchy

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.

Animation Resources

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

Color State List Resource

Define a color resources that changes based on the View state.
Saved in `res/color/` and accessed from the `R.color` class. XML files that define a state list of colors. They are saved in res/color/ and accessed from the **R.color** class.

Drawable Resources

Define various graphics with bitmaps or XML.
Saved in `res/drawable/` and accessed from the `R.drawable` class. Image files like .png, .jpg,

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

Layout Resource

Define the layout for your application UI.
Saved in `res/layout/` and accessed from the `R.layout` class. XML files that define a user interface layout. They are saved in res/layout/ and accessed from the **R.layout** class

Menu Resource

Define the contents of your application menus.
Saved in `res/menu/` and accessed from the `R.menu` class. 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.

String Resources

Define strings, string arrays, and plurals (and include string formatting and styling).
Saved in `res/values/` and accessed from the `R.string`, `R.array`, and `R.plurals` classes.

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.

Style Resource

Define the look and format for UI elements.
Saved in `res/values/` and accessed from the `R.style` class.

Font Resources

Define font families and include custom fonts in XML.
Saved in `res/font/` and accessed from the `R.font` class.

For example, here's the file hierarchy for a simple project:

```
MyProject/
    src/
        MyActivity.java
    res/
        drawable/
            graphic.png
        layout/
            main.xml
            info.xml
        mipmap/
            icon.png
        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" />
```

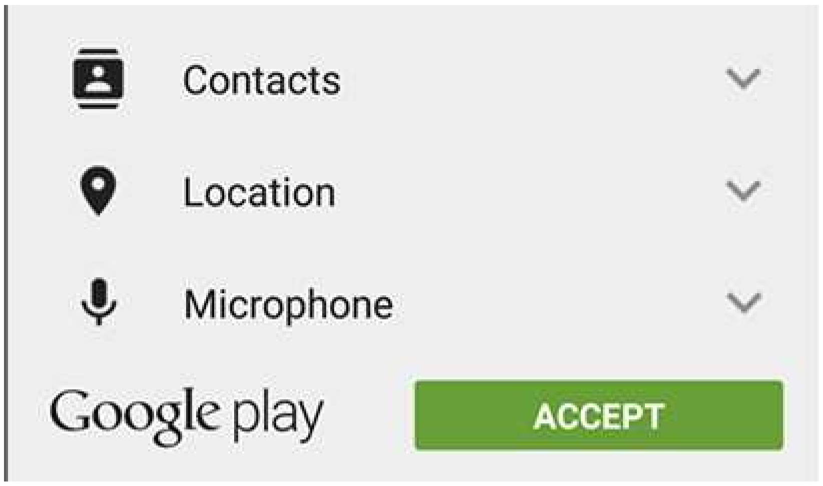
Now if you will go through previous chapter once again where I have explained **Hello World!** example, and I'm sure you will have better understanding on all the concepts explained in this chapter. So I highly recommend to check previous chapter for working example and check how I have used various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

Permissions in Android Application

Starting from **Android 6.0 (API 23)**, users are not asked for permissions at the time of installation rather developers need to request for the permissions at the run time. Only the permissions that are **defined in the manifest file** can be requested at run time.

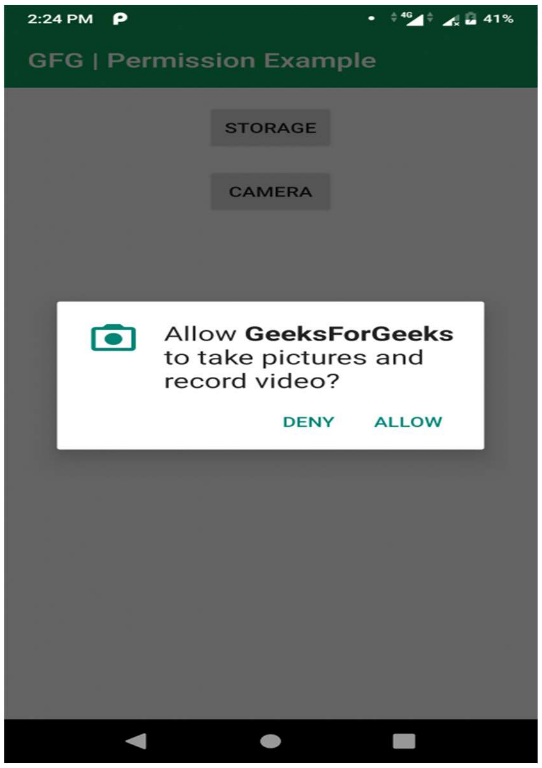
Types of Permissions:

1. **Install-Time Permissions:** If the **Android 5.1.1 (API 22) or lower**, the permission is requested at the installation time at the **Google Play Store**.



If the user **Accepts** the permissions, the app is installed. Else the app **installation is cancelled**.

2. **Run-Time Permissions:** If the **Android 6 (API 23) or higher**, the permission is requested at the run time during the runnnig of the app.



Allow **GeeksForGeeks** to take pictures and record video?

If the user **Accepts** the permissions, then that feature of the app can be used. Else to use the feature, the app **requests the permission again**.

So, now the permissions are requested at runtime. In this article, we will discuss how to request permissions in an Android Application at run time.

Steps for Requesting permissions at run time :

1. **Declare the permission in Android Manifest file:** In Android permissions are declared in **AndroidManifest.xml** file using the **uses-permission** tag.

   <uses-permission android:name=”android.permission.PERMISSION_NAME”/>

   Here we are declaring storage and camera permission.

```
<!--Declaring the required permissions-->

<uses-permission

    android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

```
<uses-permission

    android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

<uses-permission

    android:name="android.permission.CAMERA" />
```

- **Modify activity_main.xml file to Add two buttons to request permission on button click:** Permission will be checked and requested on button click. Open **activity_main.xml** file and add two buttons in it.

```
      <!--Button to request storage permission-->

       <Button

           android:id="@+id/storage"

           android:layout_width="wrap_content"

           android:layout_height="wrap_content"

           android:text="Storage"

           android:layout_marginTop="16dp"

           android:padding="8dp"

           android:layout_below="@id/toolbar"

           android:layout_centerHorizontal="true"/>


       <!--Button to request camera permission-->

       <Button

           android:id="@+id/camera"

           android:layout_width="wrap_content"

           android:layout_height="wrap_content"

           android:text="Camera"

           android:layout_marginTop="16dp"

           android:padding="8dp"

           android:layout_below="@id/storage"

           android:layout_centerHorizontal="true"/>
```

2. **Check whether permission is already granted or not. If permission isn't already granted, request user for the permission**: In order to use any service or feature, the permissions are required. Hence we have to ensure that the permissions are given for that. If not, then the permissions are requested.

   Check for permissions: Beginning with Android 6.0 (API level 23), the user has the right to revoke permissions from any app at any time, even if the app targets a lower API level. So to use the service, the app needs to check for permissions every time.

   Syntax:

```
if(ContextCompat.checkSelfPermission(thisActivity,
                      Manifest.permission.WRITE_CALENDAR)
   != PackageManager.PERMISSION_GRANTED)
{
    // Permission is not granted
}
```

   Request Permissions: When **PERMISSION_DENIED** is returned from the **checkSelfPermission()** method in the above syntax, we need to prompt the user for that permission. Android provides several methods that can be used to request permission, such as **requestPermissions()**.

   Syntax:

```
ActivityCompat.requestPermissions(MainActivity.this,
                                  permissionArray,
                                  requestCode);
```

   Here permissionArray is an array of type String.

   Example:

```
// Function to check and request permission

public void checkPermission(String permission, int requestCode)

{


    // Checking if permission is not granted

    if (ContextCompat.checkSelfPermission(

            MainActivity.this,

            permission)
```

```
        == PackageManager.PERMISSION_DENIED) {
        ActivityCompat
            .requestPermissions(
                MainActivity.this,
                new String[] { permission },
                requestCode);
    }
    else {
        Toast
            .makeText(MainActivity.this,
                      "Permission already granted",
                      Toast.LENGTH_SHORT)
            .show();
    }
}
```

This function will show a toast message if permission is already granted otherwise prompt user for permission.

3. **Override onRequestPermissionsResult() method: onRequestPermissionsResult()** is called when user grant or decline the permission. **RequestCode** is one of the parameteres of this function which is used to check user action for corresponding request. Here a toast message is shown indicating the permission and user action.

Example:

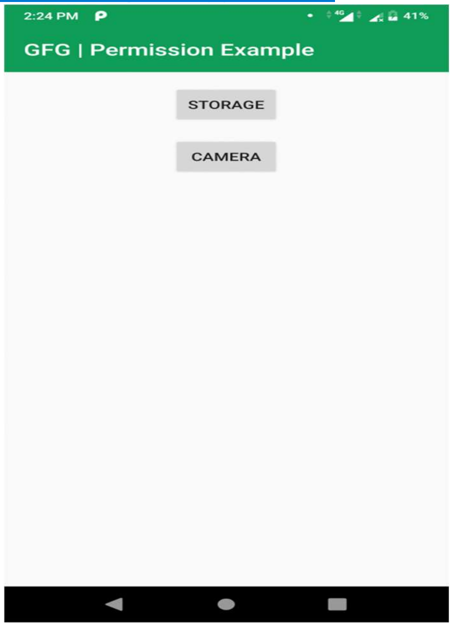
```
// This function is called when user accept or decline the permission.
// Request Code is used to check which permission called this function.
// This request code is provided when user is prompt for permission.


@Override
public void onRequestPermissionsResult(int requestCode,
                                       @NonNull String[] permissions,
                                       @NonNull int[] grantResults)
{
    super
        .onRequestPermissionsResult(requestCode,
                                    permissions,
                                    grantResults);

    if (requestCode == CAMERA_PERMISSION_CODE) {

        // Checking whether user granted the permission or not.
        if (grantResults.length > 0
            && grantResults[0] == PackageManager.PERMISSION_GRANTED) {

            // Showing the toast message
            Toast.makeText(MainActivity.this,
                           "Camera Permission Granted",
                           Toast.LENGTH_SHORT)
                .show();
        }
        else {
            Toast.makeText(MainActivity.this,
                           "Camera Permission Denied",
                           Toast.LENGTH_SHORT)
                .show();
        }
    }
    else if (requestCode == STORAGE_PERMISSION_CODE) {
        if (grantResults.length > 0
            && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
            Toast.makeText(MainActivity.this,
                           "Storage Permission Granted",
                           Toast.LENGTH_SHORT)
                .show();
        }
        else {
            Toast.makeText(MainActivity.this,
                           "Storage Permission Denied",
                           Toast.LENGTH_SHORT)
                .show();
        }
    }
}
```

Below is the complete code of this application:

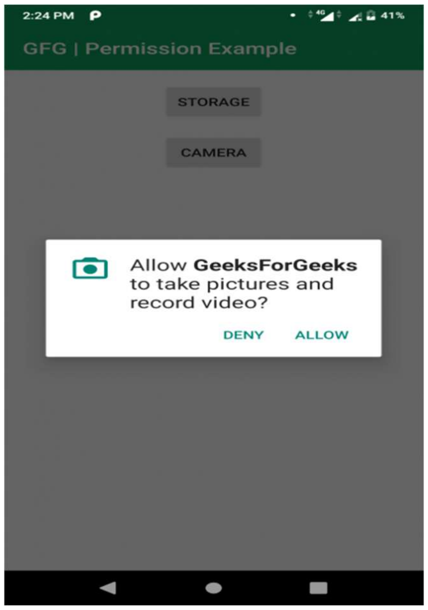
```
<?xml version="1.0" encoding="utf-8"?>
    <manifest xmlns:android="http://schemas.android.com/apk/res/android"
              package="org.geeksforgeeks.requestPermission">


        <!--Declaring the required permissions-->
        <uses-permission
            android:name="android.permission.READ_EXTERNAL_STORAGE" />
        <uses-permission
            android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
        <uses-permission
            android:name="android.permission.CAMERA" />

        <application
            android:allowBackup="true"
            android:icon="@mipmap/ic_launcher"
            android:label="@string/app_name"
            android:roundIcon="@mipmap/ic_launcher_round"
            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>
```

Output:

- **On starting the application:**

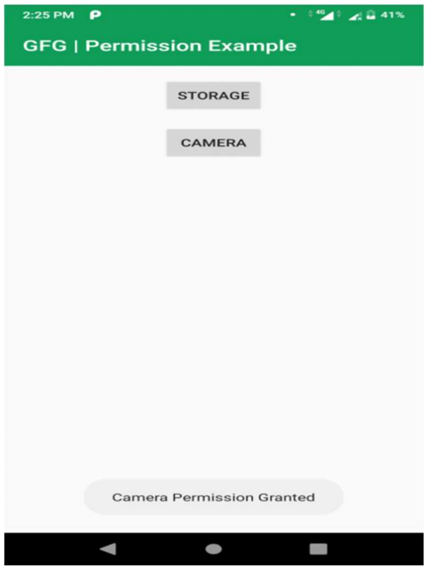


- **On clicking camera button for first time:**



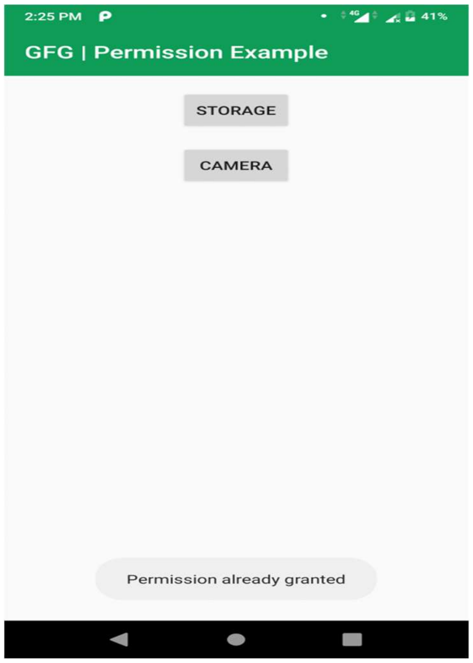
Allow **GeeksForGeeks** to take pictures and record video?

- **On Granting the permission:**



Camera Permission Granted

- **On clicking camera button again:**



Permission already granted

```
ACCESS_LOCATION_EXTRA_COMMANDS
ACCESS_NETWORK_STATE
ACCESS_NOTIFICATION_POLICY
ACCESS_WIFI_STATE
BLUETOOTH
BLUETOOTH_ADMIN
BROADCAST_STICKY
CHANGE_NETWORK_STATE
CHANGE_WIFI_MULTICAST_STATE
```

```
CHANGE_WIFI_STATE
DISABLE_KEYGUARD
EXPAND_STATUS_BAR
GET_PACKAGE_SIZE
INSTALL_SHORTCUT
INTERNET
KILL_BACKGROUND_PROCESSES
MODIFY_AUDIO_SETTINGS
NFC
READ_SYNC_SETTINGS
READ_SYNC_STATS
RECEIVE_BOOT_COMPLETED
REORDER_TASKS
REQUEST_IGNORE_BATTERY_OPTIMIZATIONS
REQUEST_INSTALL_PACKAGES
SET_ALARM
SET_TIME_ZONE
SET_WALLPAPER
SET_WALLPAPER_HINTS
TRANSMIT_IR
UNINSTALL_SHORTCUT
USE_FINGERPRINT
VIBRATE
WAKE_LOCK
WRITE_SYNC_SETTINGS
```

and

Dangerous permissions :

```
READ_CALENDAR
WRITE_CALENDAR
CAMERA
READ_CONTACTS
WRITE_CONTACTS
GET_ACCOUNTS
ACCESS_FINE_LOCATION
ACCESS_COARSE_LOCATION
RECORD_AUDIO
READ_PHONE_STATE
READ_PHONE_NUMBERS
CALL_PHONE
ANSWER_PHONE_CALLS
READ_CALL_LOG
WRITE_CALL_LOG
ADD_VOICEMAIL
USE_SIP
PROCESS_OUTGOING_CALLS
BODY_SENSORS
SEND_SMS
RECEIVE_SMS
READ_SMS
RECEIVE_WAP_PUSH
RECEIVE_MMS
READ_EXTERNAL_STORAGE
WRITE_EXTERNAL_STORAGE
```