# **Android - Overview**

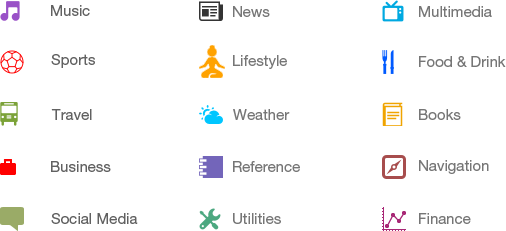
**What is Android?**

* An open source and Linux-based **Operating System** for mobile devices such as smartphones and tablet computers.
* The first beta version of the Android Software Development Kit (SDK) was released by Google in 2007 where as the first commercial version, Android 1.0, was released in September 2008.

## Features of Android

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

## Categories of Android applications



## History of Android



## What is API level?

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

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

# **Environment Setup**

First is OS:

* Microsoft® Windows® 8/7/Vista/2003 (32 or 64-bit).
* Mac® OS X® 10.8.5 or higher, up to 10.9 (Mavericks)
* GNOME or KDE desktop

Second is tools:

* Java JDK5 or later version
* Android SDK
* Java Runtime Environment (JRE) 6
* Android Studio (Best choice)
* Eclipse IDE for Java Developers (optional)
* Android Development Tools (ADT) Eclipse Plug-in (optional)

## Set-up Java Development Kit (JDK)

set PATH=C:\jdk1.7.0\_75\bin;%PATH%

set JAVA\_HOME=C:\jdk1.7.0\_75

setenv PATH /usr/local/jdk1.7.0\_75/bin:$PATH

setenv JAVA\_HOME /usr/local/jdk1.7.0\_75

# **Architecture**



## Linux kernel

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

## Libraries

On top of Linux kernel there is a set of libraries including open-source Web browser engine WebKit, well known library libc, SQLite database which is a useful repository for storage and sharing of application data, libraries to play and record audio and video, SSL libraries responsible for Internet security etc.

## Android Libraries

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

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

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

## Android Runtime

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

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

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

## Application Framework

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

The Android framework includes the following key services −

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

## Applications

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

# **Application Components**

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

**Activities**

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

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

public class MainActivity extends Activity {

}

**Services**

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

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

public class MyService extends Service {

}

**Broadcast Receivers**

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

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

public class MyReceiver extends BroadcastReceiver {

public void onReceive(context,intent){}

}

**Content Providers**

A content provider component supplies data from one application to others on request. Such requests are handled by the methods of the *ContentResolver*class. The data may be stored in the file system, the database or somewhere else entirely.

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

public class MyContentProvider extends ContentProvider {

public void onCreate(){

}

}

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

**Additional Components**

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

|  |  |
| --- | --- |
| **Components** | **Description** |
| Fragments | Represents a portion of user interface in an Activity. |
| Views | UI elements that are drawn on-screen including buttons, lists forms etc. |
| Layouts | View hierarchies that control screen format and appearance of the views. |
| Intents | Messages wiring components together. |
| Resources | External elements, such as strings, constants and drawable pictures. |
| Manifest | Configuration file for the application. |

# **Resources Organizing & Accessing**

## Organize resource in Android Studio

MyProject/

src/

main/

java/

MyActivity.java

**res/**

drawable/

icon.png

layout/

activity\_main.xml

info.xml

values/

strings.xml

|  |  |
| --- | --- |
| **Directory** | **Resource Type** |
| anim/ | XML files that define property animations. They are saved in res/anim/ folder and accessed from the **R.anim** class. |
| color/ | XML files that define a state list of colors. They are saved in res/color/ and accessed from the **R.color** class. |
| 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. |
| layout/ | XML files that define a user interface layout. They are saved in res/layout/ and accessed from the **R.layout** class. |
| 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. |
| raw/ | Arbitrary files to save in their raw form. You need to call*Resources.openRawResource()* with the resource ID, which is*R.raw.filename* to open such raw files. |
| 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. |
| xml/ | Arbitrary XML files that can be read at runtime by calling*Resources.getXML()*. You can save various configuration files here which will be used at run time. |

## Alternative Resources

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

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

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

MyProject/

src/

main/

java/

MyActivity.java

**res/**

drawable/

icon.png

background.png

**drawable-hdpi/**

icon.png

background.png

layout/

activity\_main.xml

info.xml

values/

strings.xml

## Accessing Resources

### **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);

### **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.main\_activity);

}

### **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"

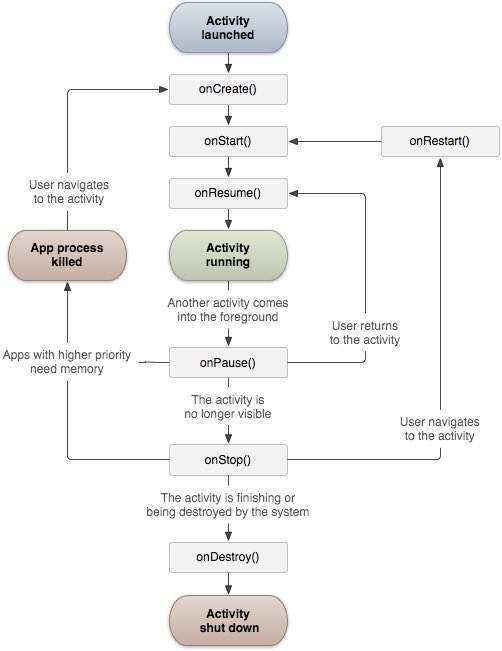
android:layout\_height="fill\_parent"

android:textColor=**"@color/opaque\_red"**

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

# **Activities**

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



|  |  |
| --- | --- |
| **Callback** | **Description** |
| onCreate() | This is the first callback and called when the activity is first created. |
| onStart() | This callback is called when the activity becomes visible to the user. |
| onResume() | This is called when the user starts interacting with the application. |
| onPause() | The paused activity does not receive user input and cannot execute any code and called when the current activity is being paused and the previous activity is being resumed. |
| onStop() | This callback is called when the activity is no longer visible. |
| onDestroy() | This callback is called before the activity is destroyed by the system. |
| onRestart() | This callback is called when the activity restarts after stopping it. |

## Example

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use eclipse IDE 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:

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

package="com.example.helloworld"

android:versionCode="1"

android:versionName="1.0" >

<uses-sdk

android:minSdkVersion="8"

android:targetSdkVersion="22" />

<application

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

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

android:theme="@style/AppTheme" >

<activity

android:name=".MainActivity"

android:label="@string/title\_activity\_main" >

<intent-filter>

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

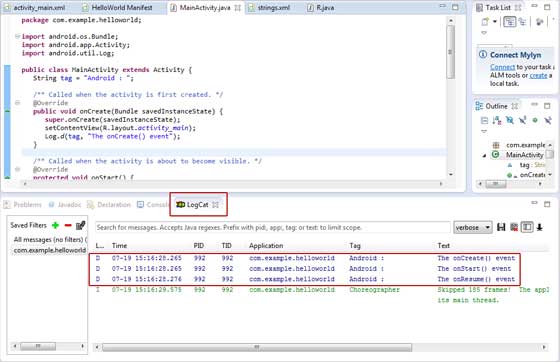
<category android:name="android.intent.category.LAUNCHER"/>

</intent-filter>

</activity>

</application>

</manifest>



 Run Eclipse Run Icon icon

07-19 15:00:43.405: D/Android :(866): The onCreate() event

07-19 15:00:43.405: D/Android :(866): The onStart() event

07-19 15:00:43.415: D/Android :(866): The onResume() event

Red button Android Red Button

07-19 15:01:10.995: D/Android :(866): The onPause() event

07-19 15:01:12.705: D/Android :(866): The onStop() event

Menu button Android Menu Button

07-19 15:01:13.995: D/Android :(866): The onStart() event

07-19 15:01:14.705: D/Android :(866): The onResume() event

Back button Android Back Button

07-19 15:33:15.687: D/Android :(992): The onPause() event

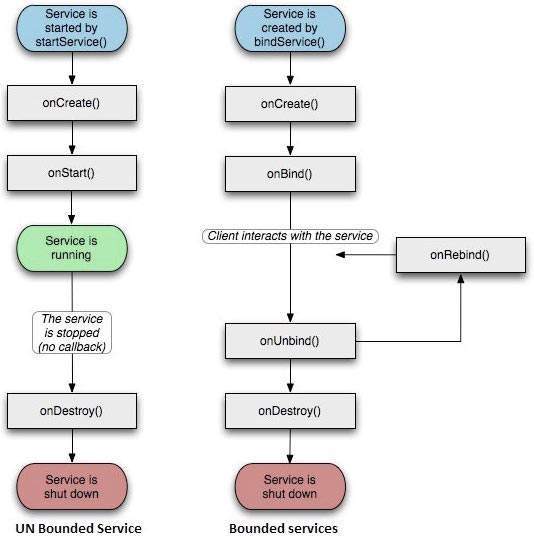
07-19 15:33:15.525: D/Android :(992): The onStop() event

07-19 15:33:15.525: D/Android :(992): The onDestroy() event

# **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

|  |  |
| --- | --- |
| **State** | **Description** |
| 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. |
| 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). |



|  |  |
| --- | --- |
| **Callback** | **Description** |
| 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. |
| 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*. |
| onUnbind() | The system calls this method when all clients have disconnected from a particular interface published by the service. |
| 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)*. |
| 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. |
| 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. |

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

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use Android StudioIDE to create an Android application and name it as *My Application* under a package *com.example.My Application* 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 **src/com.example.My Application/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.My Application;

import android.os.Bundle;

import android.app.Activity;

import android.view.Menu;

import android.content.Intent;

import android.view.View;

public class MainActivity extends Activity {

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

}

@Override

public boolean onCreateOptionsMenu(Menu menu) {

getMenuInflater().inflate(R.menu.activity\_main, menu);

return true;

}

// Method to start the service

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 **src/com.example.My Application/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.My Application;

import android.app.Service;

import android.content.Intent;

import android.os.IBinder;

import android.widget.Toast;

public class MyService extends Service {

@Override

public IBinder onBind(Intent arg0) {

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:

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

package="com.example.MyApplication"

android:versionCode="1"

android:versionName="1.0" >

<uses-sdk

android:minSdkVersion="13"

android:targetSdkVersion="22" />

<application

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

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

android:theme="@style/AppTheme" >

<activity

android:name=".MainActivity"

android:label="@string/title\_activity\_main" >

<intent-filter>

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

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

</intent-filter>

</activity>

<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>

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

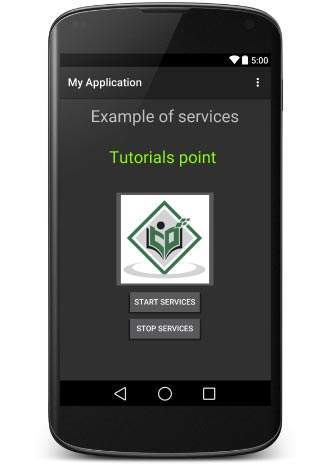
<resources>

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

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

<string name="title\_activity\_main">MainActivity</string>

</resources>



# **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

## Creating the Broadcast Receiver

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

public class MyReceiver extends BroadcastReceiver {

@Override

public void onReceive(Context context, Intent intent) {

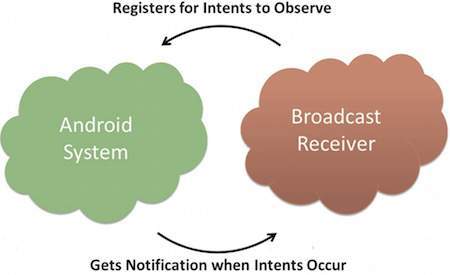
Toast.makeText(context, "Intent Detected.", Toast.LENGTH\_LONG).show();

}

}

## Registering Broadcast Receiver

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



<application

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

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

android:theme="@style/AppTheme" >

<receiver android:name="MyReceiver">

<intent-filter>

<action android:name="android.intent.action.BOOT\_COMPLETED">

</action>

</intent-filter>

</receiver>

</application>

|  |  |
| --- | --- |
| **Event Constant** | **Description** |
| android.intent.action.BATTERY\_CHANGED | Sticky broadcast containing the charging state, level, and other information about the battery. |
| android.intent.action.BATTERY\_LOW | Indicates low battery condition on the device. |
| android.intent.action.BATTERY\_OKAY | Indicates the battery is now okay after being low. |
| android.intent.action.BOOT\_COMPLETED | This is broadcast once, after the system has finished booting. |
| android.intent.action.BUG\_REPORT | Show activity for reporting a bug. |
| android.intent.action.CALL | Perform a call to someone specified by the data. |
| android.intent.action.CALL\_BUTTON | The user pressed the "call" button to go to the dialer or other appropriate UI for placing a call. |
| android.intent.action.DATE\_CHANGED | The date has changed. |
| android.intent.action.REBOOT | Have the device reboot. |

## Broadcasting Custom Intents

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

public void broadcastIntent(View view)

{

Intent intent = new Intent();

intent.setAction("com.tutorialspoint.CUSTOM\_INTENT");

sendBroadcast(intent);

}

This intent *com.tutorialspoint.CUSTOM\_INTENT* can also be registered in similar way as we have 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

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use Android studio to create an Android application and name it as*My Application* under a package *com.example.My Application* 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.My Application* to define a BroadcastReceiver. |
| 4 | An application can handle one or more custom and system intents without any restrictions. Every indent 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**src/com.example.My Application/MainActivity.java**. This file can include each of the fundamental life cycle methods. We have added *broadcastIntent()*method to broadcast a custom intent.

package com.example.My Application;

import android.os.Bundle;

import android.app.Activity;

import android.view.Menu;

import android.content.Intent;

import android.view.View;

public class MainActivity extends Activity {

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

}

@Override

public boolean onCreateOptionsMenu(Menu menu) {

getMenuInflater().inflate(R.menu.activity\_main, menu);

return true;

}

// broadcast a custom intent.

public void broadcastIntent(View view){

Intent intent = new Intent();

intent.setAction("com.tutorialspoint.CUSTOM\_INTENT");

sendBroadcast(intent);

}

}

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

package com.example.My Application;

import android.content.BroadcastReceiver;

import android.content.Context;

import android.content.Intent;

import android.widget.Toast;

public class MyReceiver extends BroadcastReceiver {

@Override

public void onReceive(Context context, Intent intent) {

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

}

}

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

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

package="com.example.My Application"

android:versionCode="1"

android:versionName="1.0" >

<uses-sdk

android:minSdkVersion="8"

android:targetSdkVersion="22" />

<application

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

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

android:theme="@style/AppTheme" >

<activity

android:name=".MainActivity"

android:label="@string/title\_activity\_main" >

<intent-filter>

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

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

</intent-filter>

</activity>

<receiver android:name="MyReceiver">

<intent-filter>

<action android:name="com.tutorialspoint.CUSTOM\_INTENT">

</action>

</intent-filter>

</receiver>

</application>

</manifest>

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

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

xmlns:tools="http://schemas.android.com/tools" android:layout\_width="match\_parent"

android:layout\_height="match\_parent" android:paddingLeft="@dimen/activity\_horizontal\_margin"

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

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

android:paddingBottom="@dimen/activity\_vertical\_margin" tools:context=".MainActivity">

<TextView

android:id="@+id/textView1"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Example of Broadcast"

android:layout\_alignParentTop="true"

android:layout\_centerHorizontal="true"

android:textSize="30dp" />

<TextView

android:id="@+id/textView2"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Tutorials point "

android:textColor="#ff87ff09"

android:textSize="30dp"

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

android:layout\_centerHorizontal="true"

android:layout\_marginBottom="40dp" />

<ImageButton

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/imageButton"

android:src="@drawable/abc"

android:layout\_centerVertical="true"

android:layout\_centerHorizontal="true" />

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/button2"

android:text="Broadcast Intent"

android:onClick="broadcastIntent"

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

android:layout\_centerHorizontal="true" />

</RelativeLayout>

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

<resources>

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

<string name="title\_activity\_main">My Application</string>

</resources>

