

Android Core Building Blocks



An android **component** is simply a piece of code that has a well defined life cycle e.g. Activity, Receiver, Service etc.

The **core building blocks** or **fundamental components** of android are activities, views, intents, services, content providers, fragments and AndroidManifest.xml.

Activity

An activity is a class that represents a single screen. It is like a Frame in AWT.

View

A view is the UI element such as button, label, text field etc. Anything that you see is a view.

Intent

Intent is used to invoke components. It is mainly used to:

- Start the service
- Launch an activity
- Display a web page
- Display a list of contacts
- Broadcast a message
- Dial a phone call etc.

For example, you may write the following code to view the webpage.

1. `Intent intent=new Intent(Intent.ACTION_VIEW);`
 2. `intent.setData(Uri.parse("http://www.javatpoint.com"));`
 3. `startActivity(intent);`
-

Service

Service is a background process that can run for a long time.

There are two types of services local and remote. Local service is accessed from within the application whereas remote service is accessed remotely from other applications running on the same device.

Content Provider

Content Providers are used to share data between the applications.

Fragment

Fragments are like parts of activity. An activity can display one or more fragments on the screen at the same time.

AndroidManifest.xml

It contains information about activities, content providers, permissions etc. It is like the web.xml file in Java EE.

Android Virtual Device (AVD)

It is used to test the android application without the need for mobile or tablet etc. It can be created in different configurations to emulate different types of real devices.

Activity in Android

In this tutorial we will learn about one of the most important concepts related to Android development, which is **Activity**.

What is an Activity in Android?

Human mind or conscious is responsible for what we feel, what we think, makes us feel pain when we are hurt(physically or emotionally), which often leads to a few tears, laughing on seeing or hearing something funny and a lot more. What happens to us in the real world physically(getting hurt, seeing, hearing etc) are interpreted by our mind(conscious or soul) and we think or operate as per.

So in a way, we can say that our body is just a physical object while what controls us through every situation is our mind(soul or conscious).

In case of Android → Views, Layouts and ViewGroups are used to design the user interface, which is the physical appearance of our App. But what is the mind or soul or conscious of our App? Yes, it is the **Activity**.

Activity is nothing but a java class in Android which has some pre-defined functions which are triggered at different App states, which we can override to perform anything we want.

Activity class provides us with empty functions allowing us to be the controller of everything.

For example, if we have a function specified by our mind → onSeeingSomethingFunny(), although we know what happens inside this, but what if we can override and provide our own definition to this function.

@Override

onSeeingSomethingFunny()

```
{  
    start crying;  
}
```

One thing that is different here in context to our example is, that a human is created once at birth, and is destroyed once at death, and for the time in between is controlled by the

mind/soul/concious. But an Activity is responsible to create and destroy an App infinite number of times. So apart from controlling the app, Activity also controls creation, destruction and other states of the App's lifecycle.

There can be multiple Activities in Android, but there can be only one Main Activity. For example, In Java programming (or programming languages like C or C++), the execution of the program always begin with main() method. Similarly, when the user presses the App icon, the Main Activity is called and the execution starts from the onCreate() method of the Activity class.

Different States of App (or, the main App Activity)

Starting from a user clicking on the App icon to launch the app, to the user exiting from the App, there are certain defined states that the App is in, let's see what they are.

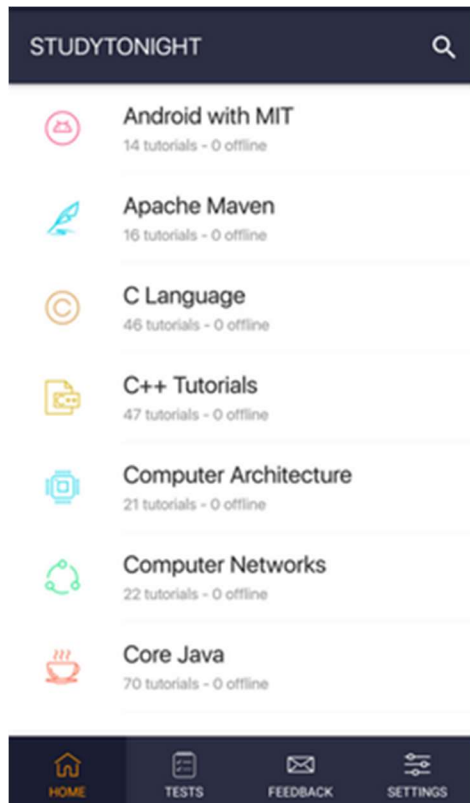
1. When a user clicks on the App icon, the Main Activity gets started and it creates the App's User Interface using the layout XMLs. And the App or Activity starts running and it is said to be in **ACTIVE** state.
2. When any dialog box appears on the screen, like when you press exit on some apps, it shows a box confirming whether you want to exit or not. At that point of time, we are not able to interact with the App's UI until we deal with that dialog box/popup. In such a situation, the Activity is said to be in **PAUSED** state.
3. When we press the Home button while using the app, our app doesn't closes. It just get minimized. This state of the App is said to be **STOPPED** state.
4. When we finally destroy the App i.e when we completely close it, then it is said to be in **DESTROYED** state.

Hence, all in all there are four states of an Activity(App) in Android namely, Active, Paused, Stopped and Destroyed.

From the user's perspective, The activity is either visible, partially visible or invisible at a given point of time. So what happens when an Activity has one of the following visibility? We will learn about that, first let's learn about these states in detail.

Active State

- When an Activity is in active state, it means it is active and running.
- It is visible to the user and the user is able to interact with it.
- Android Runtime treats the Activity in this state with the highest priority and never tries to kill it.



Activity State: Created, Started or Resumed

Process state is Foreground.

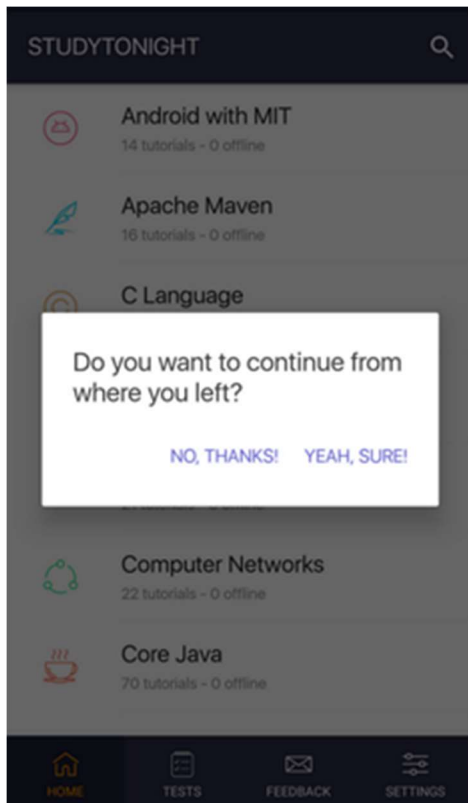
And the likelihood of killing the app is Least.

Paused State

An activity being in this state means that the user can still see the Activity in the background such as behind a transparent window or a dialog box i.e it is partially visible.

The user cannot interact with the Activity until he/she is done with the current view.

Android Runtime usually does not kill an Activity in this state but may do so in an extreme case of resource crunch.



Activity State:

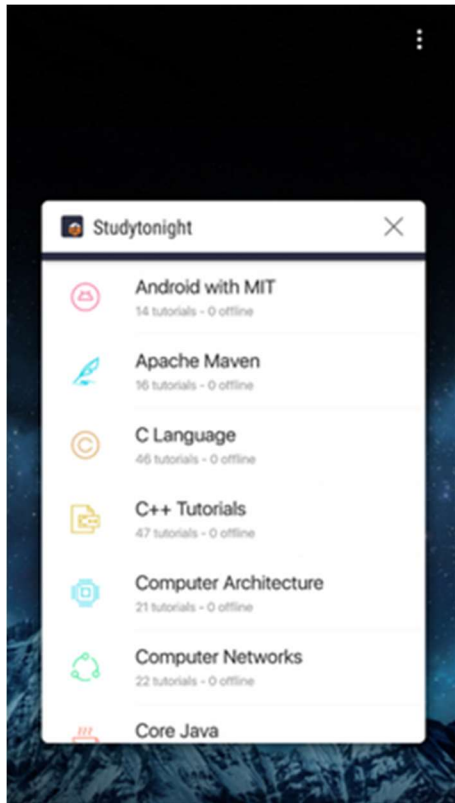
Paused

Process state is
Background(lost focus).

And the likelihood of killing the
app is More.

Stopped State

- When a new Activity is started on top of the current one or when a user hits the Home key, the activity is brought to Stopped state.
- The activity in this state is invisible, but it is not destroyed.
- Android Runtime may kill such an Activity in case of resource crunch.



Activity State:

Stopped

Process state is
Background(not visible).

And the likelihood of killing the
app is Most.

Destroyed State

- When a user hits a Back key or Android Runtime decides to reclaim the memory allocated to an Activity i.e in the paused or stopped state, It goes into the Destroyed state.
- The Activity is out of the memory and it is invisible to the user.

Note: An Activity does not have the control over managing its own state. It just goes through state transitions either due to user interaction or due to system-generated events.

Activity Lifecycle methods

Whenever we open Google Maps app, it fetches our location through GPS. And if the GPS tracker is off, then Android will ask for your permission to switch it ON. So how the GPS tracker or Android is able to decide whether an app needs GPS tracker for functioning or not?

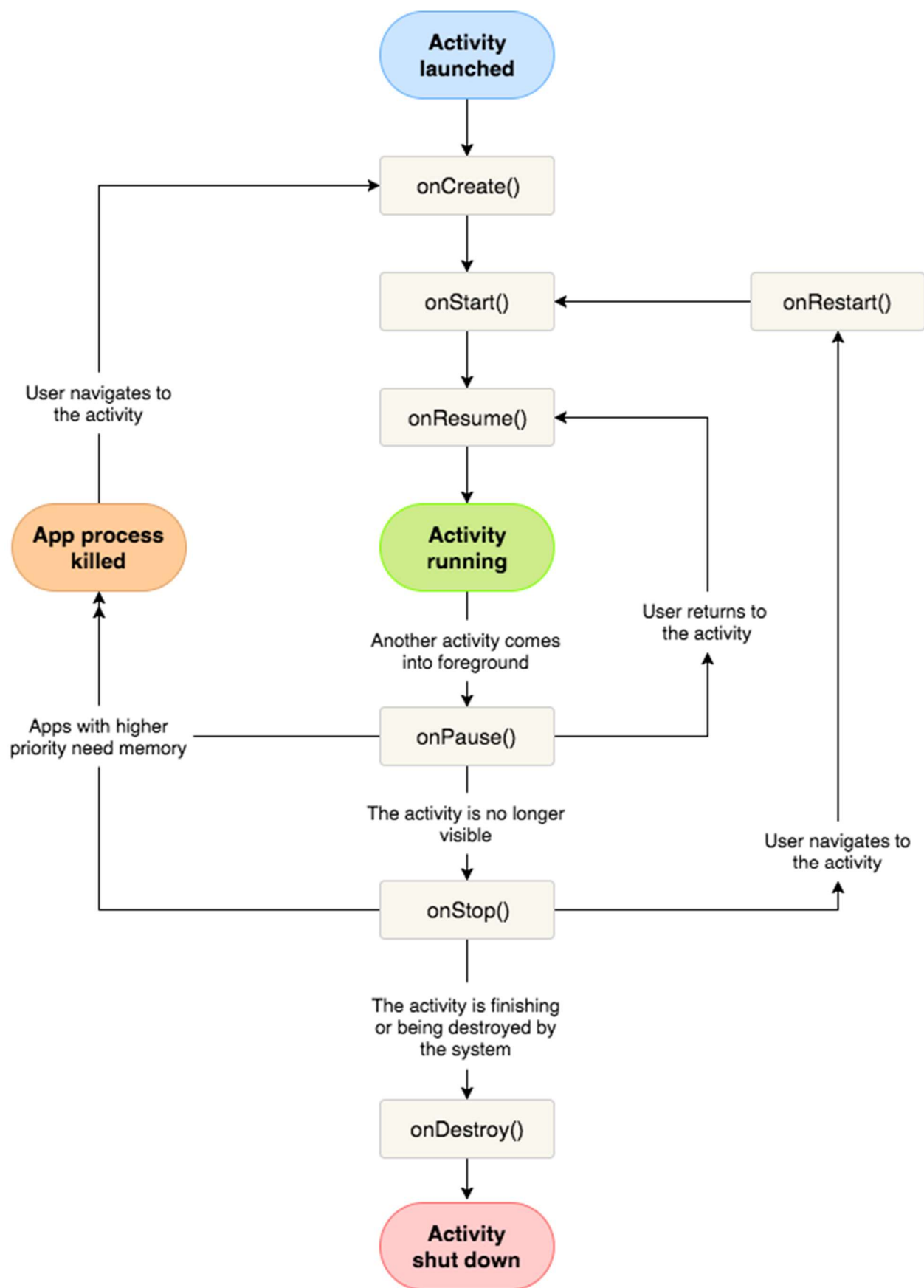
Yes, obviously, the App when started asks for the GPS location which is only possible if the GPS tracker is switched ON.

And how does the App knows this, because we coded it that whenever a user starts it, it has to ask the user to switch ON the GPS tracker, as it is required.

Similarly, we can also tell the app to perform a few things before exiting or getting destroyed.

This is the role of Activity Lifecycle. There are six key lifecycle methods through which every Activity goes depending upon its state. They are:

1. onCreate()
2. onStart()
3. onResume()
4. onPause()
5. onStop()
6. onDestroy()



Method	What does it do?
onCreate()	Whenever an Activity starts running, the first method to get executed is onCreate(). This method is executed only once during the lifetime. If we have any instance variables in the Activity, the initialization of those variables can be done in this method. After onCreate() method, the onStart() method is executed.
onStart()	During the execution of onStart() method, the Activity is not yet rendered on screen but is about to become visible to the user. In this method, we can perform any operation related to UI components.
onResume()	When the Activity finally gets rendered on the screen, onResume() method is invoked. At this point, the Activity is in the active state and is interacting with the user.
onPause()	If the activity loses its focus and is only partially visible to the user, it enters the paused state. During this transition, the onPause() method is invoked. In the onPause() method, we may commit database transactions or perform light-weight processing before the Activity goes to the background.
onStop()	From the active state, if we hit the Home key, the Activity goes to the background and the Home Screen of the device is made visible. During this event, the Activity enters the stopped state. Both onPause() and onStop() methods are executed.

When an activity is destroyed by a user or Android system, `onDestroy()` function is called.

When the Activity comes back to focus from the paused state, `onResume()` is invoked.

When we reopen any app(after pressing Home key), Activity now transits from stopped state to the active state. Thus, `onStart()` and `onResume()` methods are invoked.

Note: *`onCreate()` method is not called, as it is executed only once during the Activity life-cycle.*

To destroy the Activity on the screen, we can hit the Back key. This moves the Activity into the destroyed state. During this event, `onPause()`, `onStop()` and `onDestroy()` methods are invoked.

Note: *Whenever we change the orientation of the screen i.e from portrait to landscape or vice-versa, lifecycle methods start from the start i.e from `onCreate()` method. This is because the complete spacing and other visual appearance gets changed and adjusted.*

Processes and threads overview

When an application component starts and the application does not have any other components running, the Android system starts a new Linux process for the application with a single thread of execution. By default, all components of the same application run in the same process and thread (called the "main" thread). If an application component starts and there already exists a process for that application (because another component from the application exists), then the component is started within that process and uses the same thread of execution. However, you can arrange for different components in your application to run in separate processes, and you can create additional threads for any process.

This document discusses how processes and threads work in an Android application.

Processes

By default, all components of the same application run in the same process and most applications should not change this. However, if you find that you need to control which process a certain component belongs to, you can do so in the manifest file.

The manifest entry for each type of component element—[<activity>](#), [<service>](#), [<receiver>](#), and [<provider>](#)—supports an `android:process` attribute that can specify a process in which that component should run. You can set this attribute so that each component runs in its own process or so that some components share a process while others do not. You can also set `android:process` so that components of different applications run in the same process—provided that the applications share the same Linux user ID and are signed with the same certificates.

The [<application>](#) element also supports an `android:process` attribute, to set a default value that applies to all components.

Android might decide to shut down a process at some point, when resources are required by other processes that are more immediately serving the user. Application components running in the process that's killed are consequently destroyed. A process is started again for those components when there's again work for them to do.

When deciding which processes to kill, the Android system weighs their relative importance to the user. For example, it more readily shuts down a process hosting activities that are no longer visible on screen, compared to a process hosting visible activities. The decision whether to terminate a process, therefore, depends on the state of the components running in that process.

The details of the process lifecycle and its relationship to application states are discussed in [Processes and Application Lifecycle](#).

Threads

When an application is launched, the system creates a thread of execution for the application, called "main." This thread is very important because it is in charge of dispatching events to the appropriate user interface widgets, including drawing events. It is also almost always the thread in which your application interacts with components from the Android UI toolkit (components from the [android.widget](#) and [android.view](#) packages). As such, the main thread is also sometimes called the UI thread. However, under special circumstances, an app's main thread might not be its UI thread; for more information, see [Thread annotations](#).

The system does *not* create a separate thread for each instance of a component. All components that run in the same process are instantiated in the UI thread, and system calls to each component are dispatched from that thread. Consequently, methods that respond to system callbacks (such as [onKeyDown\(\)](#) to report user actions or a lifecycle callback method) always run in the UI thread of the process.

For instance, when the user touches a button on the screen, your app's UI thread dispatches the touch event to the widget, which in turn sets its pressed state and posts an invalidate request to the event queue. The UI thread dequeues the request and notifies the widget that it should redraw itself.

When your app performs intensive work in response to user interaction, this single thread model can yield poor performance unless you implement your application properly. Specifically, if everything is happening in the UI thread, performing long operations such as network access or database queries will block the whole UI. When the thread is blocked, no events can be dispatched, including drawing events. From the user's perspective, the application appears to hang. Even worse, if the UI thread is blocked for more than a few seconds (about 5 seconds currently) the user is presented with the infamous "[application not responding](#)" (ANR) dialog. The user might then decide to quit your application and uninstall it if they are unhappy.

Additionally, the Android UI toolkit is *not* thread-safe. So, you must not manipulate your UI from a worker thread—you must do all manipulation to your user interface from the UI thread. Thus, there are simply two rules to Android's single thread model:

1. Do not block the UI thread
2. Do not access the Android UI toolkit from outside the UI thread

Worker threads

Because of the single threaded model described above, it's vital to the responsiveness of your application's UI that you do not block the UI thread. If you have operations to perform that are not instantaneous, you should make sure to do them in separate threads ("background" or "worker" threads).

However, note that you cannot update the UI from any thread other than the UI thread or the "main" thread.

Thread

A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new Thread object, the new thread has its priority initially set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon.

When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which typically calls the method named `main` of some designated class). The Java Virtual Machine continues to execute threads until either of the following occurs:

- The exit method of class Runtime has been called and the security manager has permitted the exit operation to take place.
- All threads that are not daemon threads have died, either by returning from the call to the run method or by throwing an exception that propagates beyond the run method.

There are two ways to create a new thread of execution. One is to declare a class to be a subclass of Thread. This subclass should override the run method of class Thread. An instance of the subclass can then be allocated and started. For example, a thread that computes primes larger than a stated value could be written as follows:

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
        ...
    }
}
```

The following code would then create a thread and start it running:

```
PrimeThread p = new PrimeThread(143);
p.start();
```

The other way to create a thread is to declare a class that implements the Runnable interface. That class then implements the run method. An instance of the class can then be allocated, passed as an argument when creating Thread, and started. The same example in this other style looks like the following:

```
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
    }
}
```

```
    ...  
    }  
}
```

The following code would then create a thread and start it running:

```
PrimeRun p = new PrimeRun(143);  
new Thread(p).start();
```

Every thread has a name for identification purposes. More than one thread may have the same name. If a name is not specified when a thread is created, a new name is generated for it.

ThreadGroup

public class ThreadGroup

extends [Object](#) implements [Thread.UncaughtExceptionHandler](#)

[java.lang.Object](#)

↳ java.lang.ThreadGroup

A thread group represents a set of threads. In addition, a thread group can also include other thread groups. The thread groups form a tree in which every thread group except the initial thread group has a parent.

A thread is allowed to access information about its own thread group, but not to access information about its thread group's parent thread group or any other thread groups.

Android AsyncTask

Android AsyncTask is an abstract class provided by Android which gives us the liberty to perform heavy tasks in the background and keep the UI thread light thus making the application more responsive.

Android application runs on a single thread when launched. Due to this single thread model tasks that take longer time to fetch the response can make the application non-responsive. To avoid this we use android AsyncTask to perform the heavy tasks in background on a dedicated thread and passing the results back to the UI thread. Hence use of AsyncTask in android application keeps the UI thread responsive at all times.

The basic methods used in an android AsyncTask class are defined below :

- **doInBackground()** : This method contains the code which needs to be executed in background. In this method we can send results multiple times to the UI thread by publishProgress() method. To notify that the background processing has been completed we just need to use the return statements
- **onPreExecute()** : This method contains the code which is executed before the background processing starts
- **onPostExecute()** : This method is called after doInBackground method completes processing. Result from doInBackground is passed to this method
- **onProgressUpdate()** : This method receives progress updates from doInBackground method, which is published via publishProgress method, and this method can use this progress update to update the UI thread

The three generic types used in an android AsyncTask class are given below :

- **Params** : The type of the parameters sent to the task upon execution
- **Progress** : The type of the progress units published during the background computation
- **Result** : The type of the result of the background computation

Android AsyncTask Example

To start an AsyncTask the following snippet must be present in the MainActivity class :

```
MyTask myTask = new MyTask();  
myTask.execute();
```

In the above snippet we've used a sample classname that extends AsyncTask and execute method is used to start the background thread.

Note:

- The AsyncTask instance must be created and invoked in the UI thread.
- The methods overridden in the AsyncTask class should never be called. They're called automatically
- AsyncTask can be called only once. Executing it again will throw an exception

In this tutorial we'll implement an AsyncTask that makes a process to go to sleep for a given period of time as set by the user.

ThreadLocal

public class ThreadLocal

extends [Object](#)

[java.lang.Object](#)

↳ [java.lang.ThreadLocal<T>](#)

Known direct subclasses

[InheritableThreadLocal<T>](#)

This class provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one (via its get or set method) has its own, independently initialized copy of the variable. ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID).

For example, the class below generates unique identifiers local to each thread. A thread's id is assigned the first time it invokes ThreadId.get() and remains unchanged on subsequent calls.

```
import java.util.concurrent.atomic.AtomicInteger;
```

```
public class ThreadId {  
    // Atomic integer containing the next thread ID to be assigned  
    private static final AtomicInteger nextId = new AtomicInteger(0);  
  
    // Thread local variable containing each thread's ID
```

```

private static final ThreadLocal<Integer> threadId =
    new ThreadLocal<Integer>() {
        @Override protected Integer initialValue() {
            return nextId.getAndIncrement();
        }
    };

// Returns the current thread's unique ID, assigning it if necessary
public static int get() {
    return threadId.get();
}
}

```

Each thread holds an implicit reference to its copy of a thread-local variable as long as the thread is alive and the ThreadLocal instance is accessible; after a thread goes away, all of its copies of thread-local instances are subject to garbage collection (unless other references to these copies exist)

Android - Services

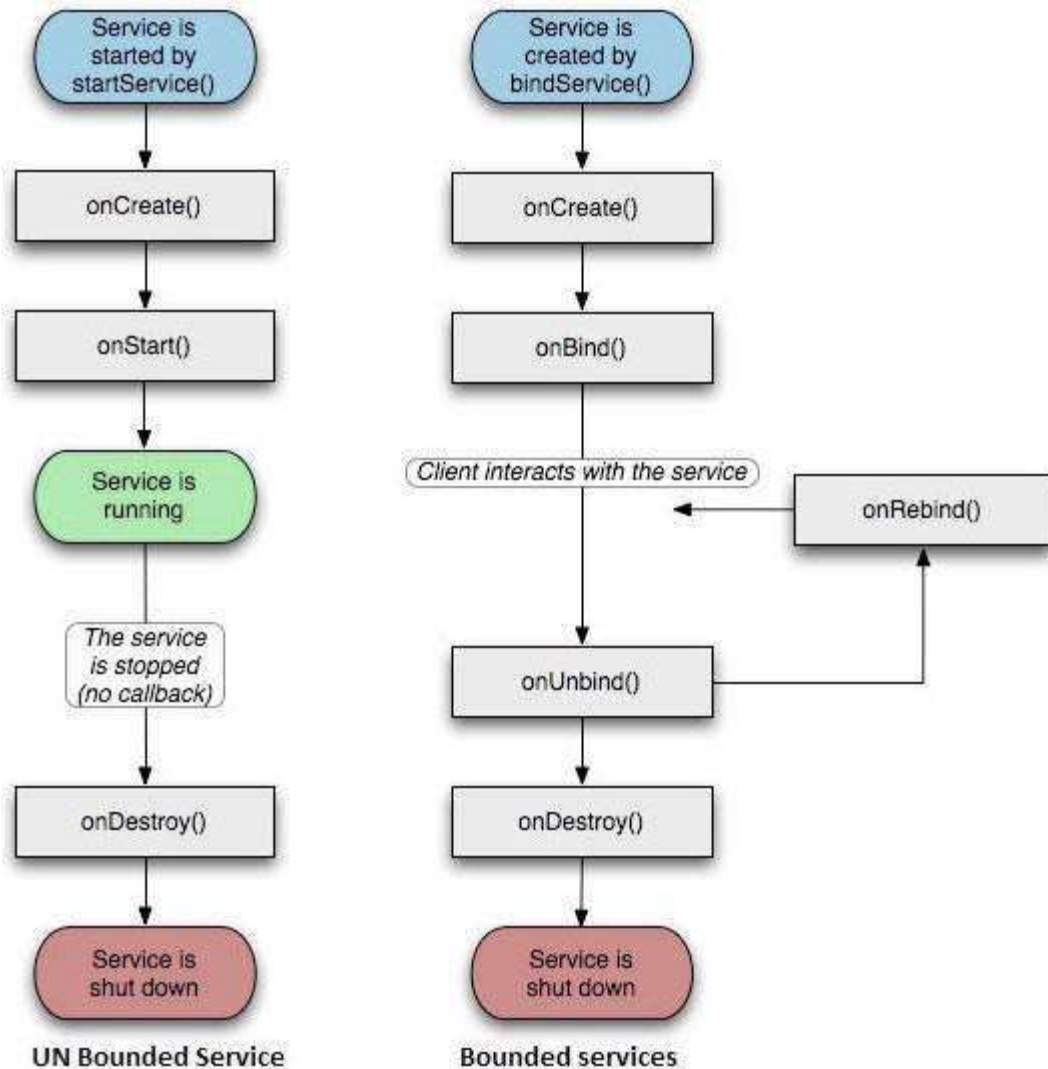
A **service** is a component that runs in the background to perform long-running operations without needing to interact with the user and it works even if application is destroyed. A service can essentially take two states –

Sr.No.	State & Description
1	<p>Started</p> <p>A service is started when an application component, such as an activity, starts it by calling <i>startService()</i>. Once started, a service can run in the background indefinitely, even if the component that started it is destroyed.</p>

Bound

A service is **bound** when an application component binds to it by calling *bindService()*. A bound service offers a client-server interface that allows components to interact with the service, send requests, get results, and even do so across processes with interprocess communication (IPC).

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



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

Sr.No.	Callback & Description
--------	------------------------

1 **onStartCommand()**

The system calls this method when another component, such as an activity, requests that the service be started, by calling *startService()*. If you implement this method, it is your responsibility to stop the service when its work is done, by calling *stopSelf()* or *stopService()* methods.

2 **onBind()**

The system calls this method when another component wants to bind with the service by calling *bindService()*. If you implement this method, you must provide an interface that clients use to communicate with the service, by returning an *IBinder* object. You must always implement this method, but if you don't want to allow binding, then you should return *null*.

3 **onUnbind()**

The system calls this method when all clients have disconnected from a particular interface published by the service.

4 **onRebind()**

The system calls this method when new clients have connected to the service, after it had previously been notified that all had disconnected in its *onUnbind(Intent)*.

5 **onCreate()**

The system calls this method when the service is first created using *onStartCommand()* or *onBind()*. This call is required to perform one-time set-up.

6 **onDestroy()**

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

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

```
package com.tutorialspoint;

import android.app.Service;
import android.os.IBinder;
import android.content.Intent;
import android.os.Bundle;

public class HelloService extends Service {

    /** indicates how to behave if the service is killed */
    int mStartMode;

    /** interface for clients that bind */
    IBinder mBinder;

    /** indicates whether onRebind should be used */
    boolean mAllowRebind;

    /** Called when the service is being created. */
    @Override
    public void onCreate() {

    }

    /** The service is starting, due to a call to startService() */
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        return mStartMode;
    }

    /** A client is binding to the service with bindService() */
    @Override
    public IBinder onBind(Intent intent) {
```

```

        return mBinder;
    }

    /** Called when all clients have unbound with unbindService() */
    @Override
    public boolean onUnbind(Intent intent) {
        return mAllowRebind;
    }

    /** Called when a client is binding to the service with bindService()*/
    @Override
    public void onRebind(Intent intent) {

    }

    /** Called when The service is no longer used and is being destroyed */
    @Override
    public void onDestroy() {

    }
}

```

Example

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

Example chapter –

Step	Description
1	You will use Android Studio IDE to create an Android application and name it as <i>My Application</i> under a package <i>com.example.tutorialspoint7.myapplication</i> as explained in the <i>Hello World Example</i> chapter.
2	Modify main activity file <i>MainActivity.java</i> to add <i>startService()</i> and <i>stopService()</i> methods.

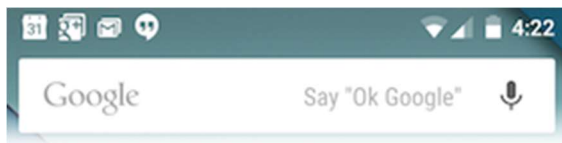
- 3 Create a new java file *MyService.java* under the package *com.example.MyApplication*. This file will have implementation of Android service related methods.
- 4 Define your service in *AndroidManifest.xml* file using `<service.../>` tag. An application can have one or more services without any restrictions.
- 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.

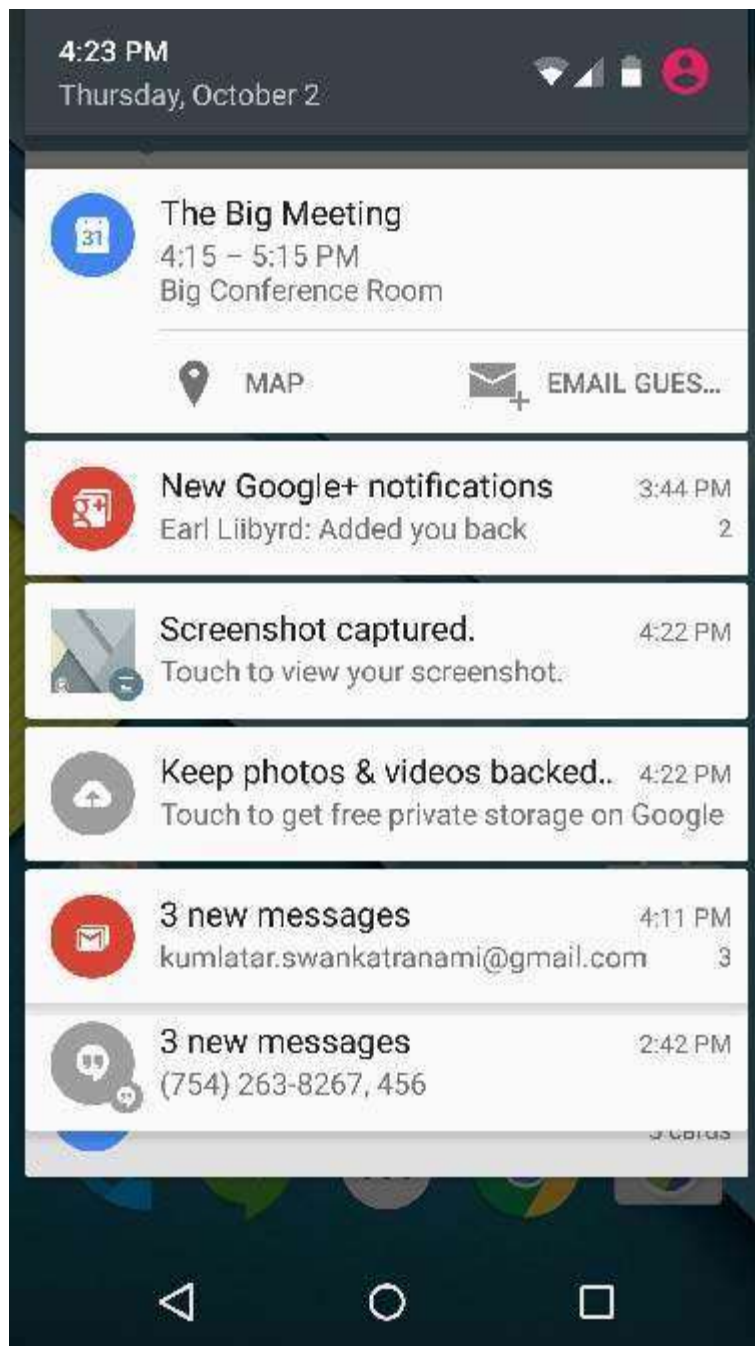
Android - Notifications

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

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



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



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

Create and Send Notifications

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

Step 1 - Create Notification Builder

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

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

Step 2 - Setting Notification Properties

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

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

```
mBuilder.setSmallIcon(R.drawable.notification_icon);  
mBuilder.setContentTitle("Notification Alert, Click Me!");  
mBuilder.setContentText("Hi, This is Android Notification Detail!");
```

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

Step 3 - Attach Actions

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

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

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

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

```
Intent resultIntent = new Intent(this, ResultActivity.class);
TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);
stackBuilder.addParentStack(ResultActivity.class);

// Adds the Intent that starts the Activity to the top of the stack
stackBuilder.addNextIntent(resultIntent);
PendingIntent resultPendingIntent =
stackBuilder.getPendingIntent(0, PendingIntent.FLAG_UPDATE_CURRENT);
mBuilder.setContentIntent(resultPendingIntent);
```

Step 4 - Issue the notification

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

```
NotificationManager mNotificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
```

```
// notificationID allows you to update the notification later on.
mNotificationManager.notify(notificationID, mBuilder.build());
```

The NotificationCompat.Builder Class

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

Sr.No.	Constants & Description
1	Notification build() Combine all of the options that have been set and return a new Notification object.

2 **NotificationCompat.Builder setAutoCancel (boolean autoCancel)**

Setting this flag will make it so the notification is automatically canceled when the user clicks it in the panel.

3 **NotificationCompat.Builder setContent (RemoteViews views)**

Supply a custom RemoteViews to use instead of the standard one.

4 **NotificationCompat.Builder setContentInfo (CharSequence info)**

Set the large text at the right-hand side of the notification.

5 **NotificationCompat.Builder setContentIntent (PendingIntent intent)**

Supply a PendingIntent to send when the notification is clicked.

6 **NotificationCompat.Builder setContentText (CharSequence text)**

Set the text (second row) of the notification, in a standard notification.

7 **NotificationCompat.Builder setTitle (CharSequence title)**

Set the text (first row) of the notification, in a standard notification.

- 8 **NotificationCompat.Builder setDefaults (int defaults)**
Set the default notification options that will be used.
- 9 **NotificationCompat.Builder setLargeIcon (Bitmap icon)**
Set the large icon that is shown in the ticker and notification.
- 10 **NotificationCompat.Builder setNumber (int number)**
Set the large number at the right-hand side of the notification.
- 11 **NotificationCompat.Builder setOngoing (boolean ongoing)**
Set whether this is an ongoing notification.
- 12 **NotificationCompat.Builder setSmallIcon (int icon)**
Set the small icon to use in the notification layouts.
- 13 **NotificationCompat.Builder setStyle (NotificationCompat.Style style)**
Add a rich notification style to be applied at build time.
- 14 **NotificationCompat.Builder setTicker (CharSequence tickerText)**
Set the text that is displayed in the status bar when the notification first arrives.
- 15 **NotificationCompat.Builder setVibrate (long[] pattern)**
Set the vibration pattern to use.

16 **NotificationCompat.Builder setWhen (long when)**

Set the time that the event occurred. Notifications in the panel are sorted by this time.

Example

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

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

- 7 Run the application to launch Android emulator and verify the result of the changes done in the application.

Following is the content of the modified main activity file

src/com.example.notificationdemo/MainActivity.java. This file can include each of the fundamental lifecycle methods.

```
package com.example.notificationdemo;

import android.app.Activity;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.content.Context;
import android.content.Intent;
import android.support.v4.app.NotificationCompat;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;

public class MainActivity extends Activity {
    Button b1;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        b1 = (Button)findViewById(R.id.button);
        b1.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                addNotification();
            }
        });
    }

    private void addNotification() {
        NotificationCompat.Builder builder =
            new NotificationCompat.Builder(this)
                .setSmallIcon(R.drawable.abc)
                .setContentTitle("Notifications Example")
                .setContentText("This is a test notification");

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



```

        PendingIntent contentIntent = PendingIntent.getActivity(this, 0, notificationIntent,
            PendingIntent.FLAG_UPDATE_CURRENT);
        builder.setContentIntent(contentIntent);

        // Add as notification
        NotificationManager manager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
        manager.notify(0, builder.build());
    }
}

```

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

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="400dp"
        android:text="Hi, Your Detailed notification view goes here...." />
</LinearLayout>

```

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

```

package com.example.notificationdemo;

import android.os.Bundle;
import android.app.Activity;

public class NotificationView extends Activity{
    @Override
    public void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        setContentView(R.layout.notification);
    }
}

```

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

```

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

```

```
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:paddingBottom="@dimen/activity_vertical_margin"
android:paddingLeft="@dimen/activity_horizontal_margin"
android:paddingRight="@dimen/activity_horizontal_margin"
android:paddingTop="@dimen/activity_vertical_margin"
tools:context="MainActivity">
```

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

```
<TextView
    android:id="@+id/textView2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Tutorials point "
    android:textColor="#ff87ff09"
    android:textSize="30dp"
    android:layout_below="@+id/textView1"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="48dp" />
```

```
<ImageButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/imageButton"
    android:src="@drawable/abc"
    android:layout_below="@+id/textView2"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="42dp" />
```

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Notification"
    android:id="@+id/button"
    android:layout_marginTop="62dp"
```

```
    android:layout_below="@+id/imageButton"
    android:layout_centerHorizontal="true" />
```

```
</RelativeLayout>
```

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

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="action_settings">Settings</string>
    <string name="app_name">tutorialspoint </string>
</resources>
```

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

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

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


        <activity
            android:name="com.example.notificationdemo.MainActivity"
            android:label="@string/app_name" >

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

        </activity>

        <activity android:name=".NotificationView"
            android:label="Details of notification"
            android:parentActivityName=".MainActivity">
            <meta-data
                android:name="android.support.PARENT_ACTIVITY"
                android:value=".MainActivity"/>
        </activity>
```

```
</application>  
</manifest>
```

Let's try to run your **tutorialspoint** application. I assume you had created your **AVD** while doing environment set-up. To run the APP from Android Studio, open one of your project's activity files and click Run  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

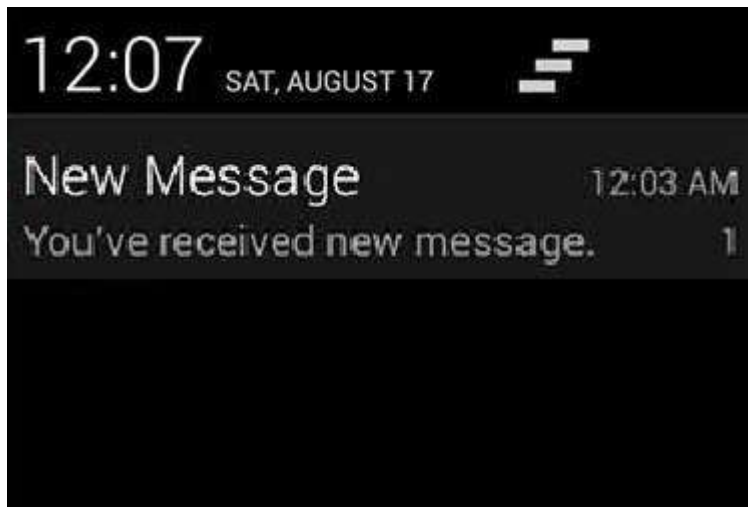
-



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

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

You will see status bar will expand and you will get following screen -



Big View Notification

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

```
protected void displayNotification() {
    Log.i("Start", "notification");

    /* Invoking the default notification service */
    NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this);

    mBuilder.setContentTitle("New Message");
    mBuilder.setContentText("You've received new message.");
    mBuilder.setTicker("New Message Alert!");
    mBuilder.setSmallIcon(R.drawable.woman);

    /* Increase notification number every time a new notification arrives */
    mBuilder.setNumber(++numMessages);

    /* Add Big View Specific Configuration */
    NotificationCompat.InboxStyle inboxStyle = new NotificationCompat.InboxStyle();

    String[] events = new String[6];
    events[0] = new String("This is first line....");
    events[1] = new String("This is second line...");
    events[2] = new String("This is third line...");
    events[3] = new String("This is 4th line...");
```

```

events[4] = new String("This is 5th line...");
events[5] = new String("This is 6th line...");

// Sets a title for the Inbox style big view
inboxStyle.setBigContentTitle("Big Title Details:");

// Moves events into the big view
for (int i=0; i < events.length; i++) {
    inboxStyle.addLine(events[i]);
}

mBuilder.setStyle(inboxStyle);

/* Creates an explicit intent for an Activity in your app */
Intent resultIntent = new Intent(this, NotificationView.class);

TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);
stackBuilder.addParentStack(NotificationView.class);

/* Adds the Intent that starts the Activity to the top of the stack */
stackBuilder.addNextIntent(resultIntent);
PendingIntent resultPendingIntent
=stackBuilder.getPendingIntent(0,PendingIntent.FLAG_UPDATE_CURRENT);

mBuilder.setContentIntent(resultPendingIntent);
mNotificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);

/* notificationID allows you to update the notification later on. */
mNotificationManager.notify(notificationID, mBuilder.build());
}

```

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

Android - Broadcast Receivers

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

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

- Creating the Broadcast Receiver.
- Registering Broadcast Receiver

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

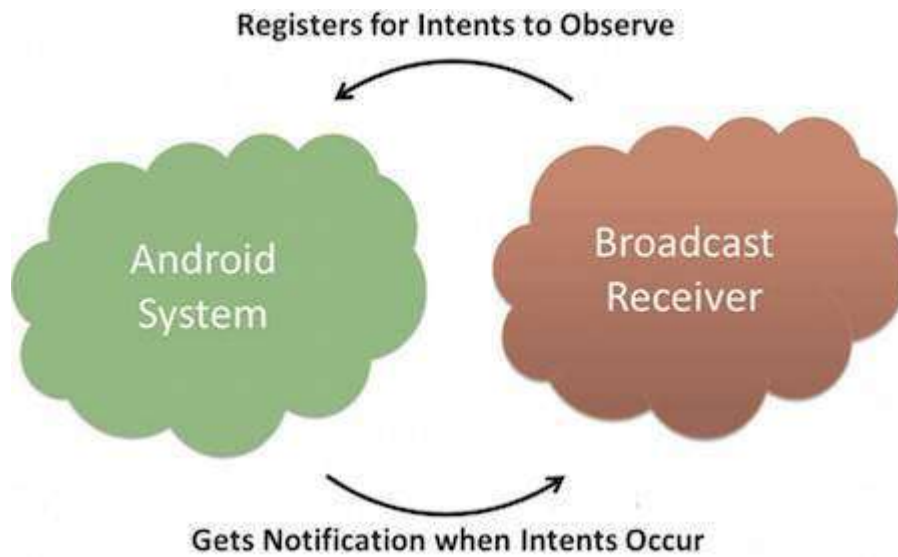
Creating the Broadcast Receiver

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

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

Registering Broadcast Receiver

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



Broadcast-Receiver

```

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

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

    </receiver>
</application>
  
```

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

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

Sr.No	Event Constant & Description
-------	------------------------------

android.intent.action.BATTERY_CHANGED

- | | |
|---|---|
| 1 | Sticky broadcast containing the charging state, level, and other information about the battery. |
|---|---|

android.intent.action.BATTERY_LOW

- 2 Indicates low battery condition on the device.

android.intent.action.BATTERY_OKAY

- 3 Indicates the battery is now okay after being low.

android.intent.action.BOOT_COMPLETED

- 4 This is broadcast once, after the system has finished booting.

android.intent.action.BUG_REPORT

- 5 Show activity for reporting a bug.

android.intent.action.CALL

- 6 Perform a call to someone specified by the data.

android.intent.action.CALL_BUTTON

- 7 The user pressed the "call" button to go to the dialer or other appropriate UI for placing a call.

android.intent.action.DATE_CHANGED

- 8 The date has changed.

android.intent.action.REBOOT

- 9 Have the device reboot.

Broadcasting Custom Intents

If you want your application itself should generate and send custom intents then you will have to create and send those intents by using the *sendBroadcast()* method inside your activity class. If

you use the *sendStickyBroadcast(Intent)* method, the Intent is **sticky**, meaning the *Intent* you are sending stays around after the broadcast is complete.

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

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

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

Example

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

Step	Description
1	You will use Android studio to create an Android application and name it as <i>My Application</i> under a package <i>com.example.tutorialspoint7.myapplication</i> as explained in the <i>Hello World Example</i> 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 be 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:supportRtl="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" />


```

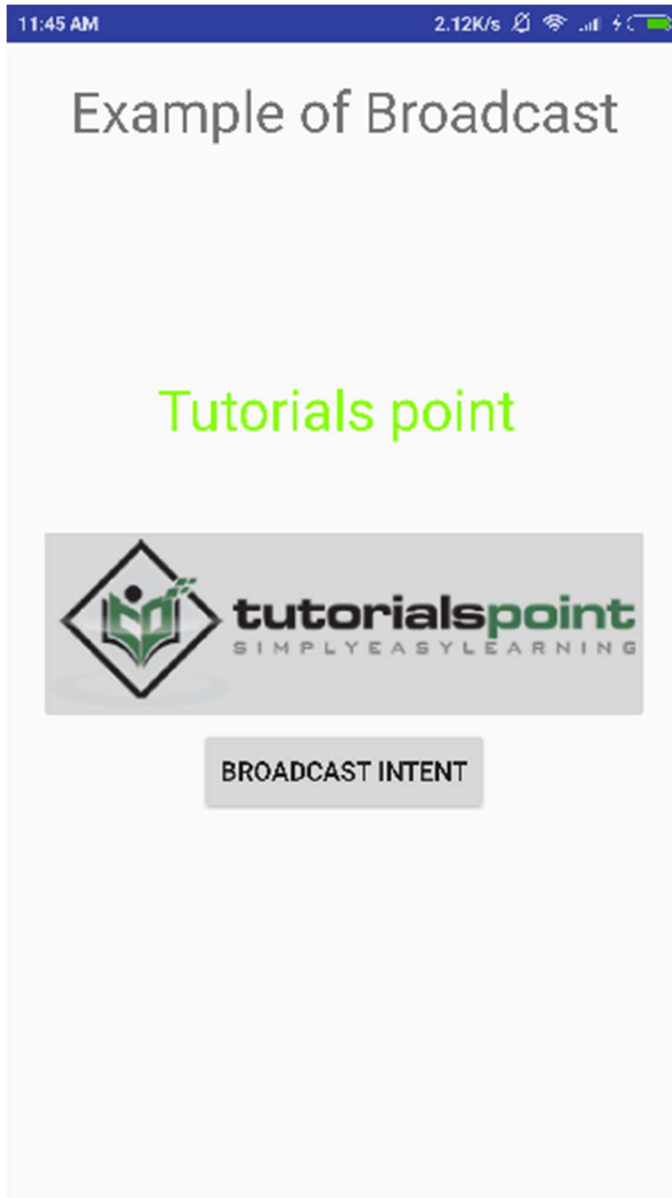
```
<TextView
    android:id="@+id/textView2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Tutorials point "
    android:textColor="#ff87ff09"
    android:textSize="30dp"
    android:layout_above="@+id/imageButton"
    android:layout_centerHorizontal="true"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

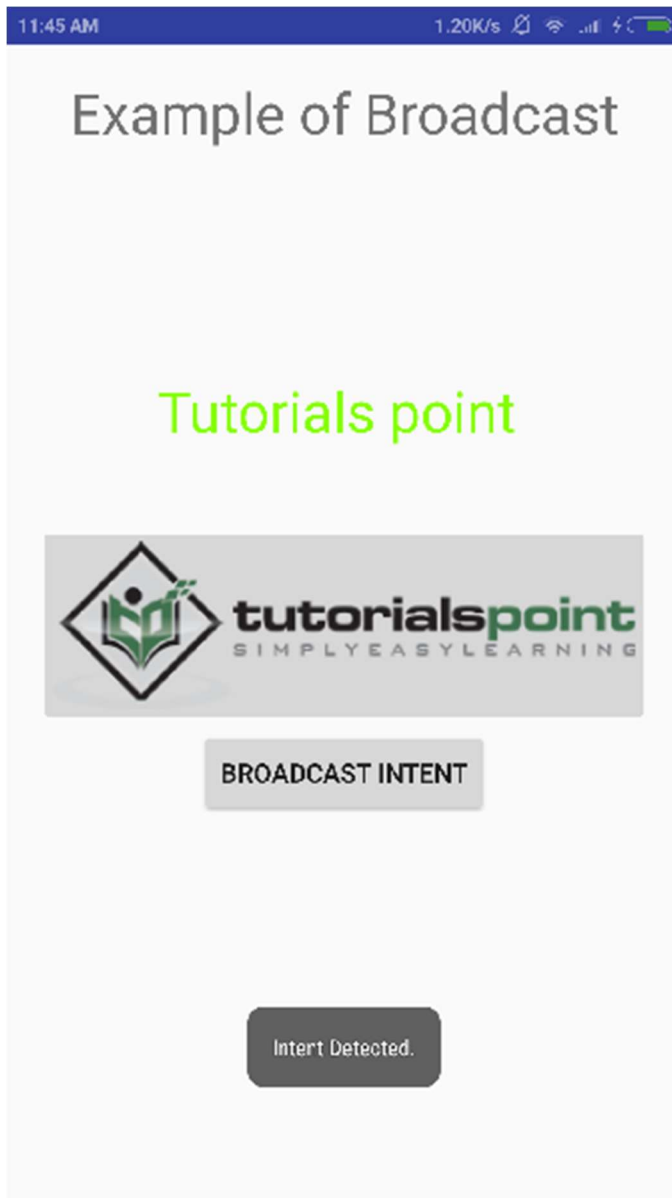
```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/button2"
    android:text="Broadcast Intent"
    android:onClick="broadcastIntent"
    android:layout_below="@+id/imageButton"
    android:layout_centerHorizontal="true" />
```

</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 –



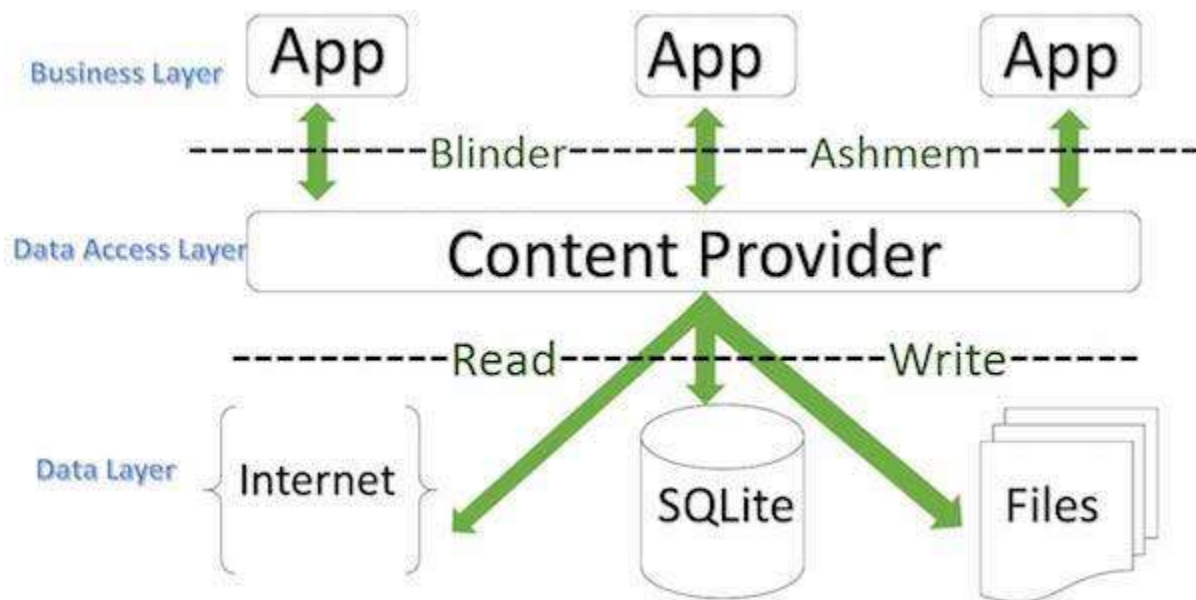
Now to broadcast our custom intent, let's click on **Broadcast Intent** button, this will broadcast our custom intent "*com.tutorialspoint.CUSTOM_INTENT*" which will be intercepted by our registered BroadcastReceiver i.e. MyReceiver and as per our implemented logic a toast will appear on the bottom of 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 - 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
	prefix
1	This is always set to content://
	authority
2	This specifies the name of the content provider, for example <i>contacts</i> , <i>browser</i> etc. For third-party content providers, this could be the fully qualified name, such as <i>com.tutorialspoint.statusprovider</i>
	data_type
3	This indicates the type of data that this particular provider provides. For example, if you are getting all the contacts from the <i>Contacts</i> content provider, then the data path would be <i>people</i> and URI would look like this <i>content://contacts/people</i>
	id
4	This specifies the specific record requested. For example, if you are looking for contact number 5 in the <i>Contacts</i> content provider then URI would look like this <i>content://contacts/people/5</i> .

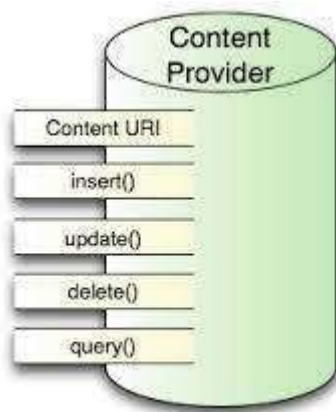
Create Content Provider

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

- First of all you need to create a Content Provider class that extends the *ContentProviderbase* class.

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

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



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 Studio IDE 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.