MODULE 3

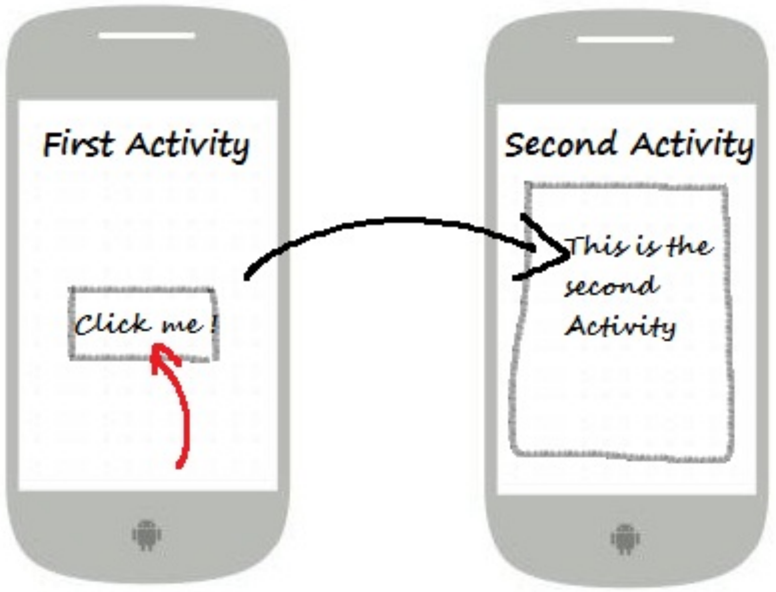
ACTIVITY

- An activity represents a single screen with a user interface just like window or frame of java that allows users to interact with the application.

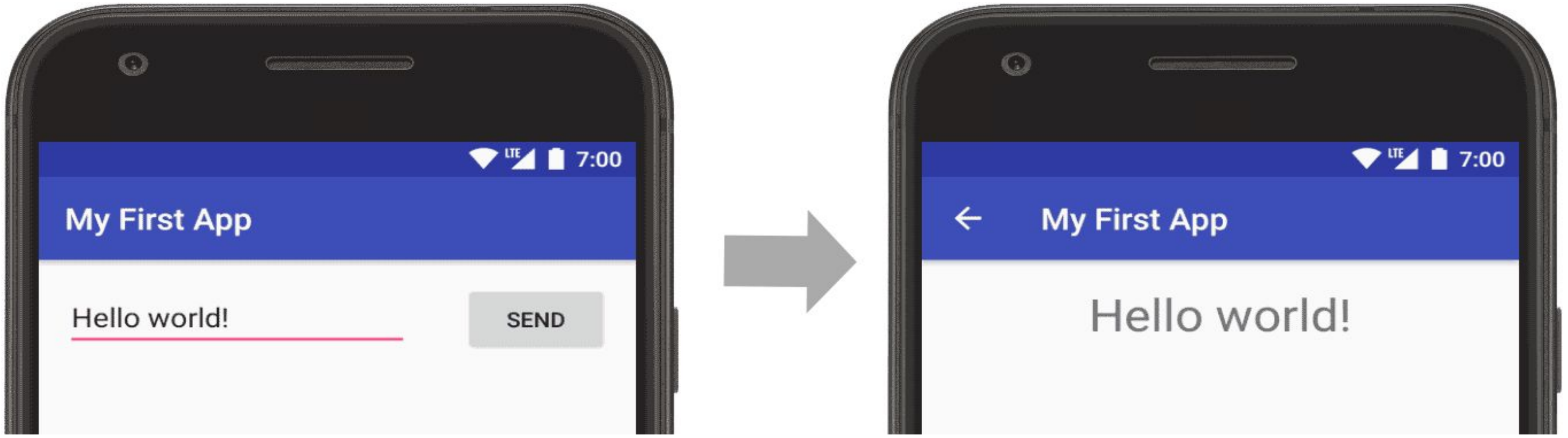
- Android activity is the subclass of **ContextThemeWrappe**r class.

- Depending on the application, there can be single to multiple activities, meaning one or more screens. Android apps start from a main activity and from there additional activities can be opened.

Example-Facebook app provides an activity to log into the application. There could be other activities like posting a photograph, send a message to a friend etc



Example of an activity

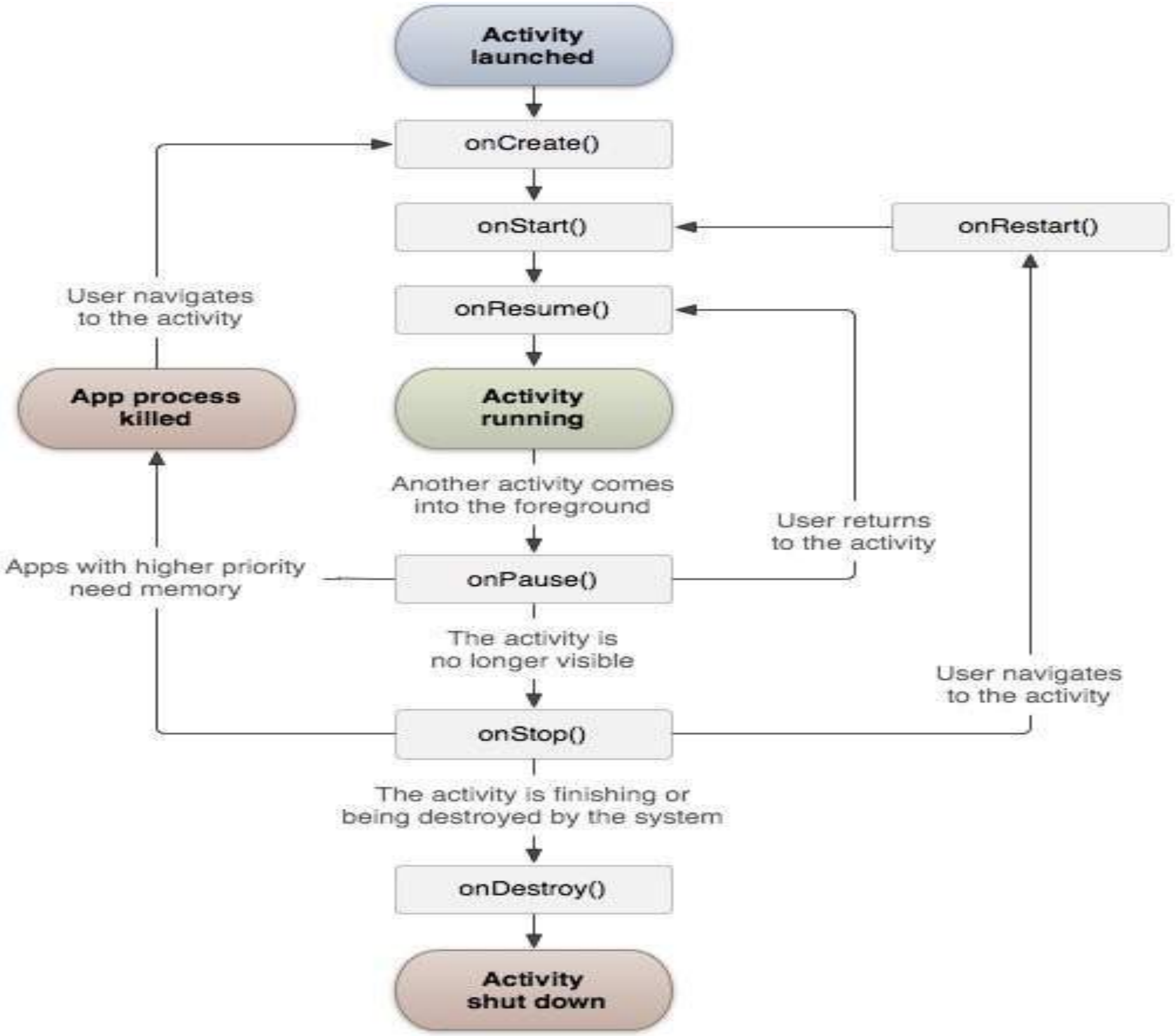


2 methods to implement Activity

- **onCreate(Bundle)**-method in which we initialize our activity. Here we usually call **setContentView(int)** with a layout resource defining the UI and **findVewbyId(id)**to retrieve the widgets in that UI that we need to interact with programmatially.
- **onPause()**-another method in which we deal with the user leaving the activity. Any changes made by the user at this point be committed.

There is a sequence of callback methods that start up an activity and a sequence of callback methods that tear down an activity as shown in the below

Activity life cycle diagram


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

1.onCreate()

First call back method which is fired when the system creates an activity for the first time.
During the activity creation, activity enters into created state.

```
@override

protected void onCreate(Bundle savedInstanceState)
{ super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
}
```

Once this method execution is finished, the activity will enter into started state and the system calls the onStart() method.

2.onStart()

This method will invoke when an activity enters into started state by completing the onCreate() method.
The onStart() method will make an activity visible to the user and this method execution will finish very quickly.

```
@override
protected void onStart()
{super.onStart();
}
```

After this method execution, activity enters into resumed state and system invokes onResume() method.

3.onResume()

When an activity enters resumed state system invokes onResume() callback method. Here the user can see the functionality and designing part of an app on a single screen.
The app will stay in resumed state, until another activity happens to take focus away from the app like getting a phone call or screen turned off.
In case of any interruptions, activity will enter into paused state. And will invoke onPause() method.

```
@override
public void onResume()
{super.onResume();
If(mcamera==null)
{initializeCamera();
}
}
```

4.onPause()
When the user leaves an activity or current activity is being paused, system invokes **onPause()** method**.**
It is used to pause activities like stop playing the music when the activity is in paused state

```
public void onPause()
{
super.onPause();
If(mcamera!=null)
{mcamera.release();
mcamera=null;
}
}
```

After the execution of onPause(),next method is either onStop() or onResume() depending on what happens after an activity enters paused state.

5.onStop()

This is invoked when an activity is no longer visible to the user. this method releases all the app resources no longer needed by the user.it can happen if current activity entered into resumed state or newly launched activity covers complete screen or it is destroyed.

Example

```
@override
protected void onStop()
{super.onStop();
}
```

The next callback method is either onRestart(),in case if the activity is coming back to interact with the user or onDestroy() in case if the activity finished running.

6.onRestart()

This method is invoked when an activity restarts itself
after stopping it.
This method is always followed by onStart() method.

7.onDestroy()
The system will invoke onDestroy() before an activity is destroyed and this is the final callback method received by the android activity.
This happens when either the activity is finished or the system destroyed the activity to save space.

```
@override
public void onDestroy()
{
super.onDestroy();
}
```
This method releases all the resources which are not released by previous onStop() method.

- **When you open the app it goes through below states**

onCreate()->onStart()->onResume()

- **When you press the back button and exit the app the sequence of states are as follows**

onPaused->onStop()->onDestroy()

- **When you press the home button, below is the sequence of states**

onPaused->onStop()

- **After pressing the home button, again you open the app from a recent task list, states will be as follows**

onRestart()->onStart()->onResume()

- **After dismissing the dialog or back button from the dialog the state is given below**

onResume()

- **If a phone is ringing and user is using the app, states are given below**

onPause()->onResume()

- **After the call ends the state is given below**

onResume()

- **When your phone screen is off below is the sequence of events**

onPaused()->onStop()

- **When your phone screen is turned back on, following activities occur**

onRestart()->onStart()->onResume()

INTENT

- To perform an action on the screen.
- Intents are message passing mechanism to communicate an action. Example-view a message, send a email, display a photograph, play a video etc.

- **Intents are very close to an activity**.
- Example-You are accessing Facebook homepage in your mobile phone and want to view a recent photograph posted by one of your friends in your homepage. To do so, we must click the photograph which results in firing the intent associated with click activity. This will lead to a new screen displaying photograph along with the associated information.

Intents provide communication between components.
3 fundamental use cases for using intents

1.Starting an activity
- An Activity represents a single screen in an app. You can start a new instance of an Activity by passing an Intent to startActivity().
- The Intent describes the activity to start and carries any necessary data.

2.Starting a service
- A Service is a component that performs operations in the background without a user interface.
- You can start a service to perform a one-time operation (such as downloading a file) by passing an Intent to startService().

3.Delivering a broadcast-
- A broadcast is a message that any app can receive. The system delivers various broadcasts for system events, such as **when the system boots up or the device starts charging.**
- You can deliver a broadcast to other apps by passing
an intent to sendBroadcast() or sendOrderedBroadcast().

INTENT STRUCTURE

- **ACTION**- general action to be performed such as ACTION_VIEW,ACTION_EDIT,ACTION_MAIN etc
- **DATA**

Data to operate on such as a person record in the contact database ,expressed as a URI(uniform resource identifier).
URI (string of characters used to identify a resource)identifies a resource either by location, or a name or both.

Some examples of action/data pairs are:
ACTION_VIEW ***content://contacts/people/1*** -- Display information about the person whose identifier is "1".
ACTION_EDIT ***content://contacts/people/1*** -- Edit information about the person whose identifier is "1".

Types of intents

❖ Explicit intent

Explicit intents specify which application will satisfy the intent, by supplying either the target app's package name or a fully-qualified component class name.
For example, you might start a new activity within your app in response to a user action, or start a service to download a file in the background.

It is used when an activity calls another activity. example-we have two activities-login and homepage activity, after login it takes user to homepage.

❖ Implicit intent

Implicit intents do not name a specific component, but instead declare a general action to perform, which allows a component from another app to handle it.

Here the target name is not passed in the intent at the time of creating it. Android itself decides which component of which application should receive this intent.

For example, if you want to show the user a location on a map, you can use an implicit intent to request that another capable app show a specified location on a map.

Intent Filter

When you use an implicit intent, the Android system finds the appropriate component to start by comparing the contents of the intent to the *intent filters* declared in the manifest file of other apps on the device. If the intent matches an intent filter, the system starts that component and delivers it the Intent object.

An intent filter is an expression in an app's manifest file that specifies the type of intents that the component would like to receive.

For instance, by declaring an intent filter for an activity, you make it possible for other apps to directly start your activity with a certain kind of intent.

Likewise, if you do *not* declare any intent filters for an activity, then it can be started only with an explicit intent.

Attributes

syntax:

```
<intent-filter android:icon="drawable resource"
               android:label="string resource"
               android:priority="integer" >
    . . .
</intent-filter>
```

- android:icon- An icon that represents the parent activity, service, or broadcast receiver when that component is presented to the user as having the capability described by the filter.
- android:label-

  The label should be set as a reference to a string resource android:priority -The priority that should be given to the parent component with regard to handling intents of the type described by the filter.

When an intent could be handled by multiple activities with different priorities, **Android will consider only those with higher priority values as potential targets for the intent.**

It controls the order in which broadcast receivers are executed to receive broadcast messages.

Those with higher priority values are called before those with lower values

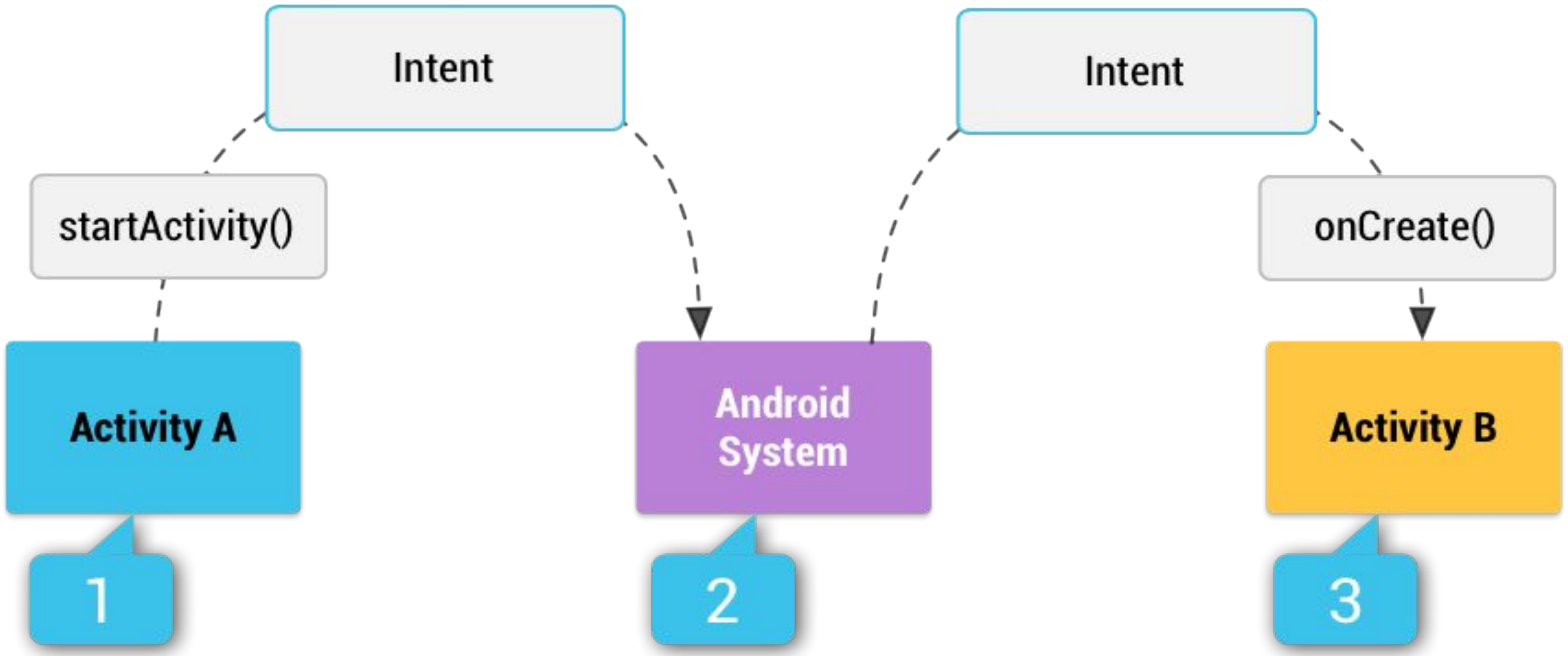


Fig shows how an implicit intent is delivered through the system to start another activity:
 [1] *Activity A* creates an Intent with an action description and passes it to startActivity()
 [2] The Android System searches all apps for an intent filter that matches the intent.
When a match is found, **[3]** the system starts the matching activity (*Activity B*) by invoking its onCreate() method and passing it the Intent.

Pending intent

- specifies an action to take in the future

- The main differences between a pendingIntent and regular intent is pendingIntent will perform at a later time where Normal/Regular intent starts immediately.

- A pendingIntent wraps a regular intent that can be passed to a foreign application (like AlarmManager or NotificationManager etc..) where you are granting that application the right to perform the action. Normally a pendingIntent is used with AlarmManager or NotificationManager.

- For example, I send a PendingIntent to AlarmManager to give AlarmManager permissions to launch an activity in my own application in 5 mins.

A pendingIntent uses the following method to handle different types of intent like
PendingIntent.getActivity() — This will start an Activity like calling context.startActivity()
PendingIntent.getBroadcast() — This will perform a Broadcast like calling context.sendBroadcast()
PendingIntent.getService() — This will start a Services like calling context.startService()

- First of all you need to create an intent with required information for creating pendingIntent.
- Then the created pendingIntent will be passed to a target application (like AlarmManager, NotificationManager etc) using target application services. Then the target application will send the intent existing in pendingIntent by calling send on pendingIntent.

```
//Creating a regular intent
Intent intent = new Intent(this, SomeActivity.class);
// Creating a pendingIntent and wrapping our intent
```

<u>syntax</u>

```
PendingIntent.getActivity(Context context, int requestCode, Intent intent, int flags)
```

Example-

```
PendingIntent pendingIntent = PendingIntent.getActivity(this, 1, intent,
PendingIntent.FLAG_UPDATE_CURRENT)
```

Flags

FLAG_CANCEL_CURRENT - If the described PendingIntent already exists, the current one should be canceled before generating a new one.

FLAG_NO_CREATE - If the described PendingIntent **does not already exist**, then simply return **null** instead of creating it.

FLAG_ONE_SHOT - PendingIntent can be used only once.

FLAG_UPDATE_CURRENT - If the described PendingIntent already exists, then keep it but replace its extra data with what is in this new Intent.

BROADCAST LIFECYCLE

- **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.**

Two important steps to make **BroadcastReceiver** works for the system broadcasted intents −

- **Creating the Broadcast Receiver.**
- **Registering Broadcast Receiver**

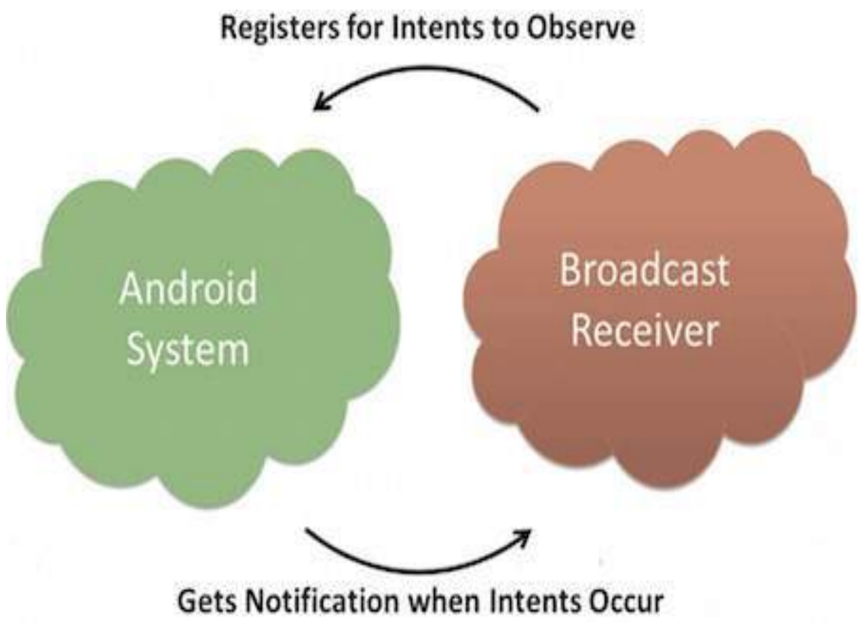
Creating the Broadcast Receiver

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.


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

```
<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.

Android BroadcastReceiver is a dormant component of android that listens to system-wide broadcast events or intents

android.intent.action.BATTERY_LOW : Indicates low battery condition on the device.
android.intent.action.BOOT_COMPLETED : This is broadcast once, after the system has finished booting
android.intent.action.REBOOT-Have the device reboot.
android.intent.action.BATTERY_OKAY
Indicates the battery is now okay after being low.

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

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.

Bounded

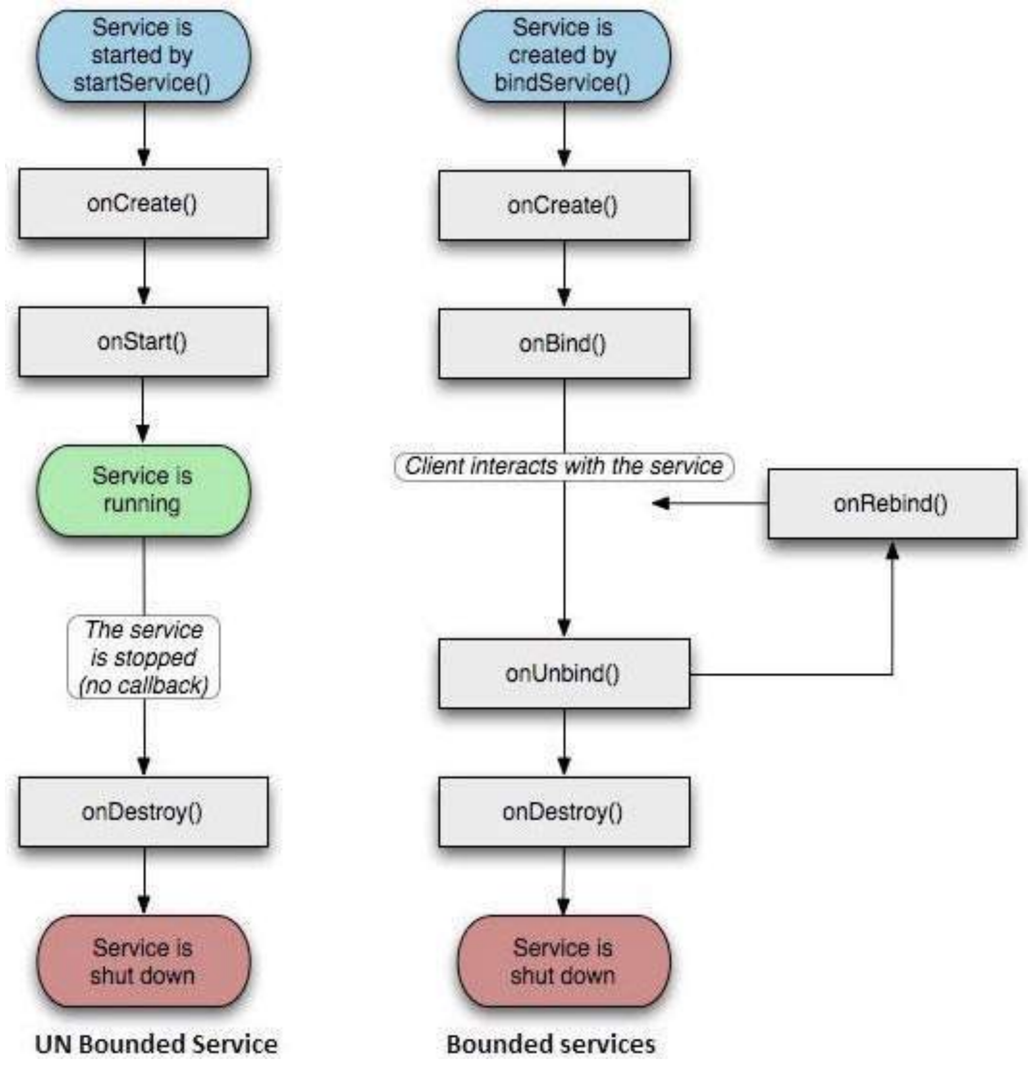
- **Bound Service-** A service is *bound* when an application component binds to it by calling *bindService()*.A bound service offers a client-server interface that allows components to interact with the service, send requests, receive results, and even do so across processes with interprocess communication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

These are the three different types of services:

Foreground service- A foreground service performs some operation that is noticeable to the user.
For example, an audio app would use a foreground service to play an audio track. Foreground services must display a notification. Foreground services continue running even when the user isn't interacting with the app.

Background service -A background service performs an operation that isn't directly noticed by the user. For example, if an app used a service to compact its storage, that would usually be a background service.

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():



1.onStartCommand()
The system calls this method when another component, such as an activity, requests that the service be started, by calling *startService()*. Once we 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.

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() by a new component*. 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.

MULTIMEDIA

The Android multimedia framework includes support for playing variety of common media types, so that you can easily integrate audio, video and images into your applications.

The following multimedia formats are supported:
Bitmap
JPEG
PNG(portable network graphics)
OGG
MPEG4
3GPP(3rd generation partnership project)
MP3

ARCHITECTURE OF ANDROID SYSTEM

Android software stack is categorized into 5 parts.
1.Linux kernel
2.Native libraries(middleware)
3.Android runtime
4.Application framework
5.Applications

1.Linux kernel -it is the heart of android architecture that interfaces computers hardware and its processes.
Responsible for device drivers, power management, memory management, device management and resource access.

2.Native libraries
On the top of linux kernel,there are native libraries like WebKit, OpenGl,FreeType,SQLite, Media,C runtime library(libc)

3.Android runtime

There are core libraries and DVM(Dalvic virtual machine) responsible to run android application.
DVM is like JVM but optimized for mobile devices. It consumes less memory and provides fast performance.

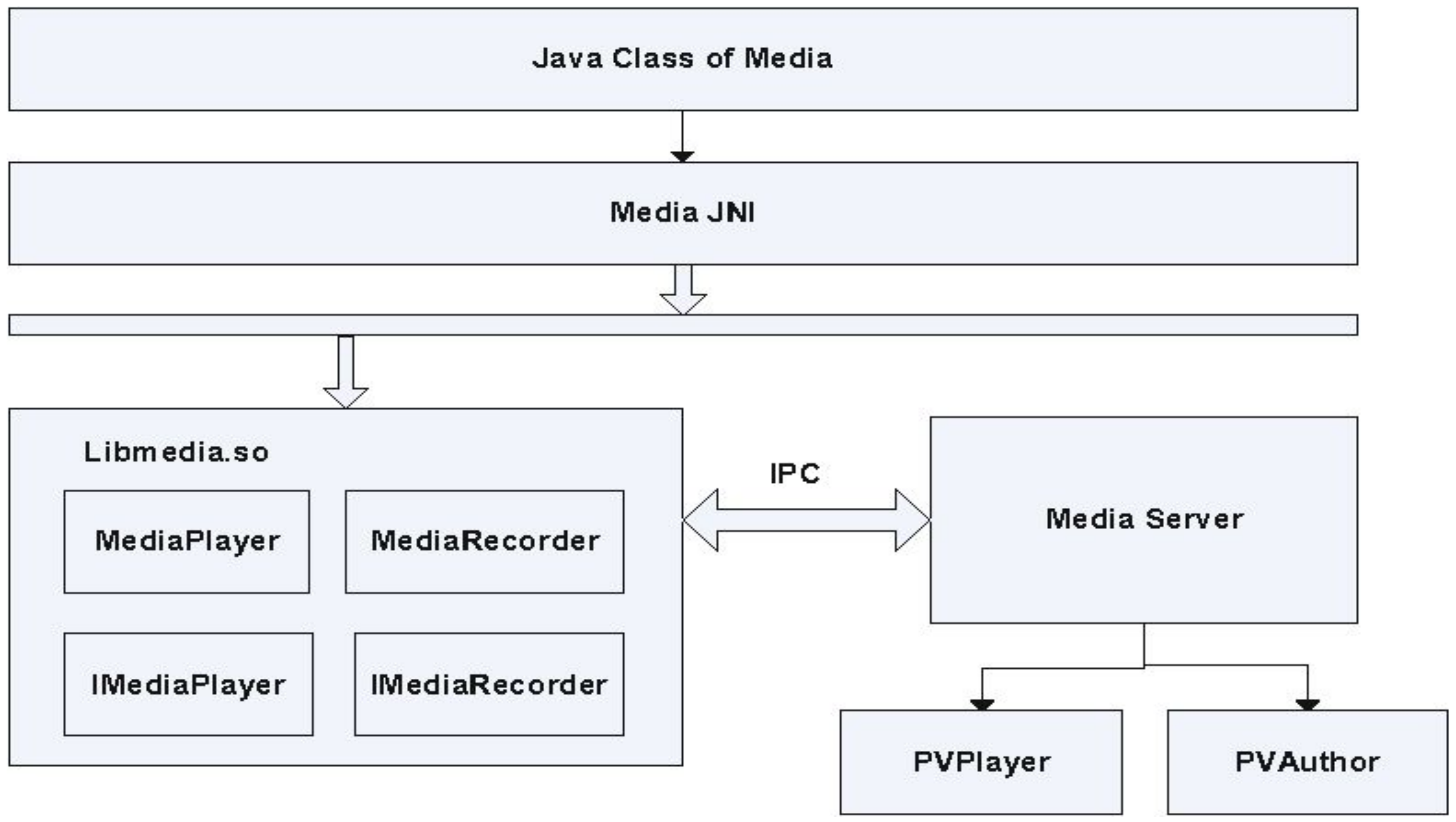
4.Android framework

Includes android API’s such as UI, telephony, resources ,locations,content providers and package managers.
Provides lot of classes and interfaces for android application development.

5 Applications

All applications such as home ,contact,settings,games,browsers are using android framework that uses android runtime and libraries. Android runtime and native libraries use linux kernel.

MULTIMEDIA FRAMEWORK



- Multimedia framework is a software framework that handles media on a computer and through a network.

- A good multimedia framework offers **attractive API and a modular architecture** to add support for new audio ,video,container formats and transmission protocols.

- Used by applications like media player, audio or video editors,also to build videoconferencing applications ,media converters and other multimedia tools.

- Java classes calls the native c library Libmedia through java native interface.

- Libmedia library communicates with media server guard process through android’s binder IPC mechanism.

Media codec/format ,container and network protocol supported by android platform.

1.Container-audio file format for storing digital audio data on a system.
2.Audio format- any format or codec can be used including ones provided by android or those that are specific to devices
3.Network protocol- the following protocols are supported in audio and videoplayback

- RTSP-real time streaming protocol(RTP-real time transport protocol,SDP-Session Description protocol)
- HTTP/HTTPS progressive streaming
- HTTP/HTTPS live streaming draft protocol
  - MPEG-2 TS media files
  - Protocol version 3(android 4 and above)

2types of classes to play audio and video in android

1.Android MediaPlayer Class
In android, by using **MediaPlayer** class we can access audio or video files from application (raw) resources, standalone files in file system or from a data stream arriving over a network connection and play audio or video files with the multiple playback options such as play, pause, forward, backward, etc.

Code snippet, to play an audio that is available in our application’s local raw resource (**res**/**raw**) directory.

```
MediaPlayer mPlayer = MediaPlayer.create(this, R.raw.songname);
mPlayer.start();
```

The second parameter in **create()** method is the name of the song that we want to play from our application resource directory (**res/raw**). In case if **raw** folder not exists in your application, create a new **raw** folder under **res** directory and add a proper encoded and formatted media files in it.

if we want to play an audio from a **URI** that is locally available in the system, we need to write the code like as shown below.

```
Uri myUri = ....; // initialize Uri here
MediaPlayer mPlayer = new MediaPlayer();
mPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);
mPlayer.setDataSource(getApplicationContext(), myUri);
mPlayer.prepare();
mPlayer.start();
```

If we want to play an audio from a **URL** via HTTP streaming, we need to write the code like as shown below.

```
String url = "http://........"; // your URL here
MediaPlayer mPlayer = new MediaPlayer();
mPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);
mPlayer.setDataSource(url);
mPlayer.prepare(); // might take long! (for buffering, etc)
mPlayer.start();
‘
```

2.AudioManager-manages audio sources, audio output on a device

MediaPlayer class provides a different type of methods to control audio and video files based on requirements.

Method	Description
getCurrentPosition()	It is used to get the current position of the song in milliseconds.
getDuration()	It is used to get the total time duration of the song in milliseconds.
isPlaying()	It returns true / false to indicate whether song playing or not.
pause()	It is used to pause the song playing.
setAudioStreamType()	it is used to specify the audio streaming type.
setDataSource()	It is used to specify the path of audio / video file to play.
setVolume()	It is used to adjust media player volume either up / down.
seekTo(position)	It is used to move song to particular position in milliseconds.
getTrackInfo()	It returns an array of track information.
start()	It is used to start playing the audio/video.
stop()	It is used to stop playing the audio/video.
reset()	It is used to reset the MediaPlayer object.
release()	It is used to releases the resources which are associated with MediaPlayer object.

TEXT TO SPEECH IN ANDROID

Android allows you convert your text into voice. Not only you can convert it but it also allows you to speak text in variety of different languages.

Android provides **TextToSpeech** class for this purpose. In order to use this class, you need to instantiate an object of this class and also specify the **initListener**.

Its syntax is given below –

```
private EditText et;
ttobj=new TextToSpeech(getApplicationContext(), new TextToSpeech.OnInitListener()
{ @Override
public void OnInit(int status)
 { } });
```

In this listener, you have to specify the properties for TextToSpeech object , such as its language ,pitch e.t.c. Language can be set by calling **setLanguage()** method. Its syntax is given below –

```
ttobj.setLanguage(Locale.UK);
```

The method setLanguage takes an Locale object as parameter. The list of some of the locales available are given below

Sr.No	Locale
1	US
2	CANADA_FRENCH
3	GERMANY
4	ITALY
5	JAPAN
6	CHINA

Once you have set the language, you can call **speak** method of the class to speak the text. Its syntax is given below –

```
ttobj.speak(toSpeak, TextToSpeech.QUEUE_FLUSH, null);
```

Some other methods available in the TextToSpeech class

1	**addSpeech(String text, String filename)** This method adds a mapping between a string of text and a sound file.
2	**getLanguage()** This method returns a Locale instance describing the language.
3	**isSpeaking()** This method checks whether the TextToSpeech engine is busy speaking.
4	**setPitch(float pitch)** This method sets the speech pitch for the TextToSpeech engine.
5	**setSpeechRate(float speechRate)** This method sets the speech rate.
6	**shutdown()** This method releases the resources used by the TextToSpeech engine.
7	**stop()** This method stop the speak.