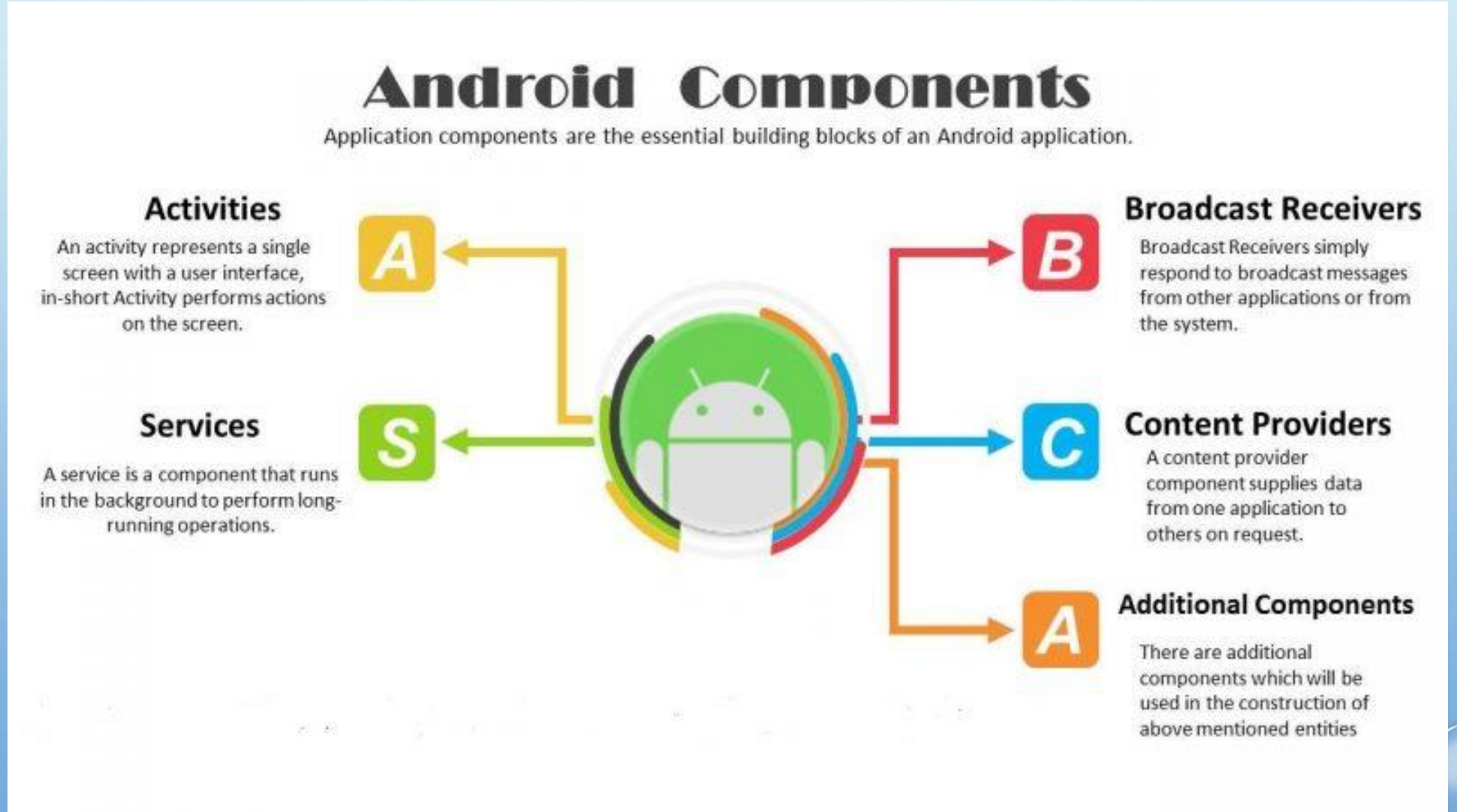
BUILDING BLOCKS OF ANDROID

INTRODUCTION



ACTIVITIES

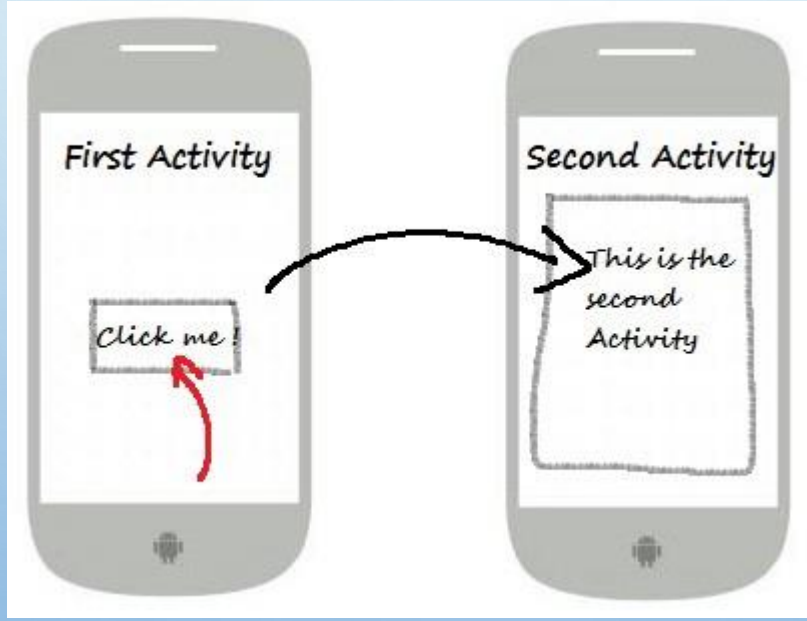
- The Activity class is a crucial component of an Android app, and the way activities are launched and put together is a fundamental part of the platform's application model.
- Unlike programming paradigms in which apps are launched with a main() method, the Android system initiates code in an Activity instance by invoking specific callback methods that correspond to specific stages of its lifecycle.

ACTIVITIES

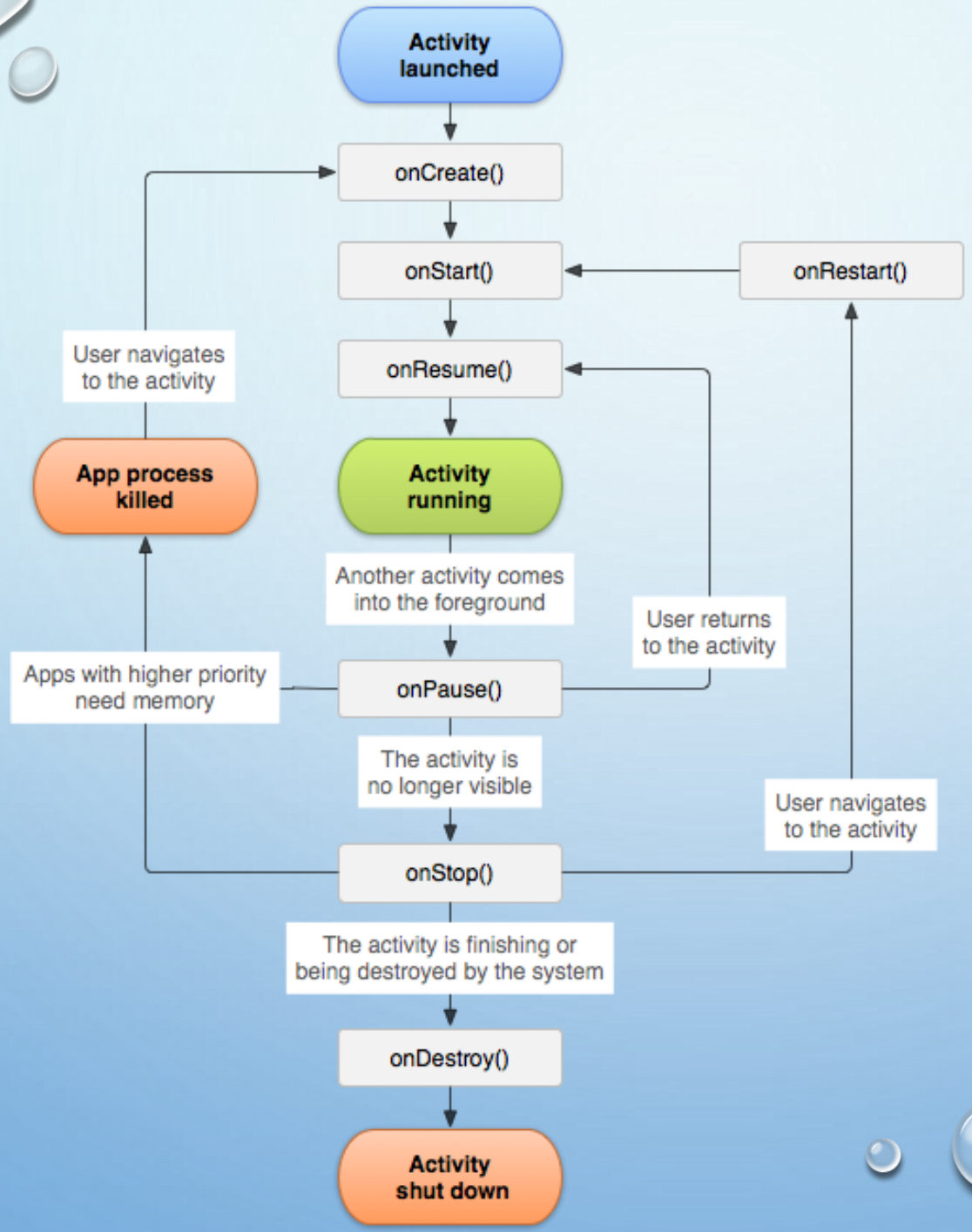
- An activity represents a single screen with a user interface.
- If an application has more than one activity, then one of them should be marked as the activity that is presented when the application is launched.

ACTIVITIES

- An activity represents a single screen with a user interface.
- If an application has more than one activity, then one of them should be marked as the activity that is presented when the application is launched.



ACTIVITY LIFE CYCLE



ACTIVITY LIFE CYCLE

- If general programming languages we see that programs start from **main()** function.
- Very similar way, Android system initiates its program with in an **Activity** starting with a call on *onCreate()* callback method.
- There is a sequence of callback methods that start up an activity and a sequence of callback methods that tear down an activity as shown in the Activity life cycle diagram
- Depending on the complexity of your activity, you probably don't need to implement all the lifecycle methods.
- However, it's important that you understand each one and implement those that ensure your app behaves the way users expect.

ON CREATE

- You MUST implement this callback, which fires when the system first creates the activity.
- On activity creation, the activity enters the Created state.
- In the onCreate() method, you perform basic application startup logic that should happen **only once for the entire life of the activity**.
- Your activity does not reside in the Created state. After the onCreate() method finishes execution, the activity enters the Started state, and the system calls the onStart() and onResume() methods in quick succession.

ON START

- When the activity enters the Started state, the system invokes this callback.
- The onStart() call makes the activity visible to the user, as the app prepares for the activity to enter the foreground and become interactive.
- The onStart() method completes very quickly and, as with the Created state, the activity does not stay resident in the Started state.
- Once this callback finishes, the activity enters the Resumed state, and the system invokes the onResume() method.

ON RESUME

- When the activity enters the Resumed state, it comes to the foreground, and then the system invokes the onResume() callback.
- This is the state in which the app interacts with the user.
- The app stays in this state until something happens to take focus away from the app.
- Such an event might be, for instance, receiving a phone call, the user's navigating to another activity, or the device screen's turning off.

ON PAUSE

- The system calls this method as the first indication that the user is leaving your activity, though it does not always mean the activity is being destroyed.
- It indicates that the activity is no longer in the foreground (though it may still be visible if the user is in multi-window mode).
- There are several reasons why an activity may enter this state. For example:
  - Some event interrupts app execution, as described in the onResume() section. This is the most common case.
  - In Android 7.0 (API level 24) or higher, multiple apps run in multi-window mode. Because only one of the apps (windows) has focus at any time, the system pauses all of the other apps.
  - A new, semi-transparent activity (such as a dialog) opens. As long as the activity is still partially visible but not in focus, it remains paused.

ON STOP

- When your activity is no longer visible to the user, it has entered the Stopped state, and the system invokes the onStop() callback.
- This may occur, for example, when a newly launched activity covers the entire screen.
- The system may also call onStop() when the activity has finished running, and is about to be terminated.
- In the onStop() method, the app should release or adjust resources that are not needed while the app is not visible to the user.
- For example, if you can't find a more opportune time to save information to a database, you might do so during onStop()

ON DESTROY

- It is called before the activity is destroyed. The system invokes this callback either because:
- The activity is finishing (due to the user completely dismissing the activity or due to finish() being called on the activity), or
- The system is temporarily destroying the activity due to a configuration change (such as device rotation or multi-window mode)
- The onDestroy() callback should release all resources that have not yet been released by earlier callbacks such as onStop().

ON RESTART

- From the Stopped state, the activity either comes back to interact with the user, or the activity is finished running and goes away.
- If the activity comes back, the system invokes onRestart().
- It fetches the instance of the activity already residing in the system memory and make it visible to the user.

SERVICES

Deep dive into

Android Services

SERVICES

- A Service is an application component that can perform long-running operations in the background, and it doesn't provide a user interface.
- Another application component can start a service, and it continues to run in the background even if the user switches to another application.
- Additionally, a component can bind to a service to interact with it and even perform inter process communication (IPC).
- For example, a service can handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

SERVICES

- These are the three different types of services:

1. Foreground

- 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.

SERVICES

2. Background

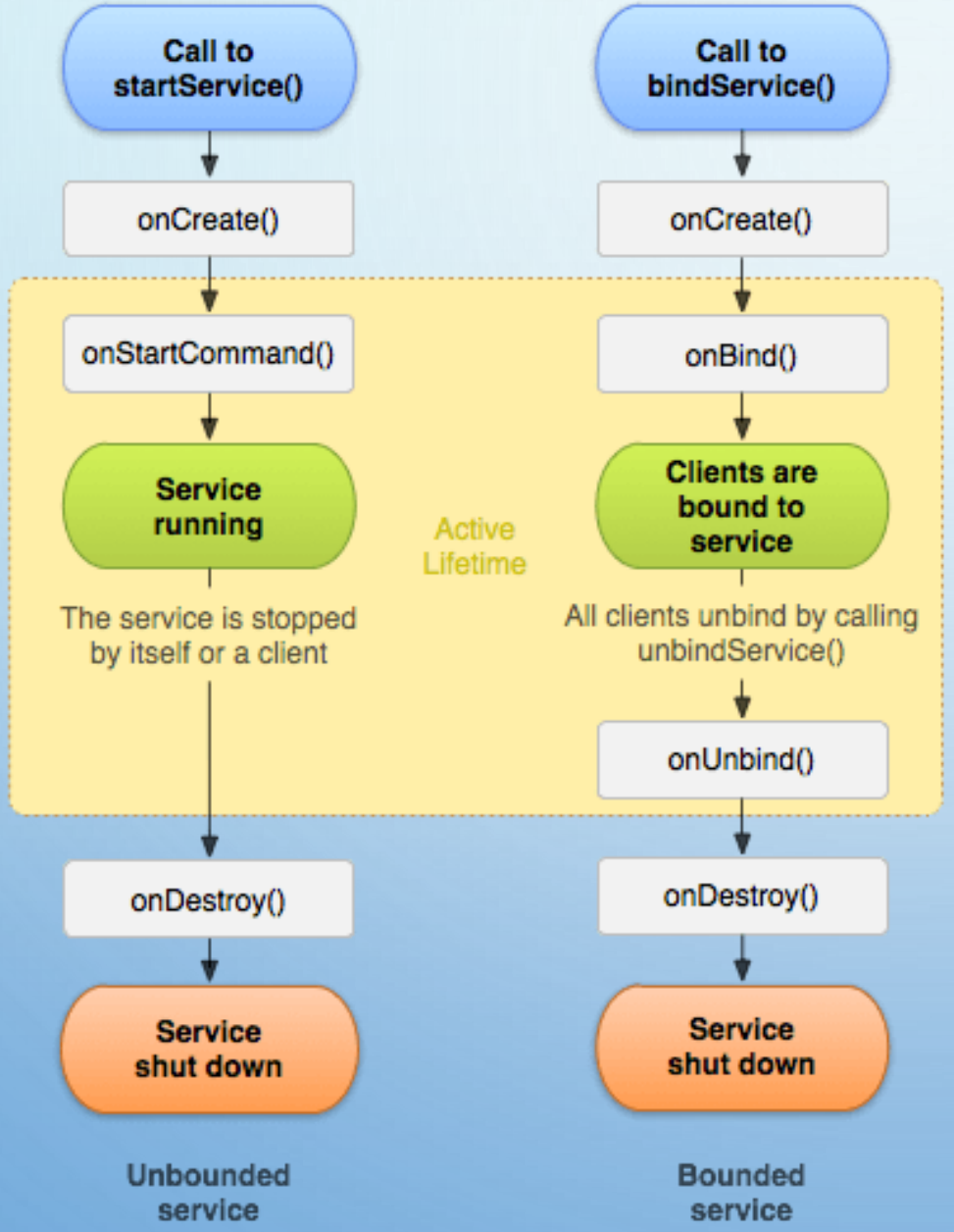
- A background service performs an operation that isn't directly noticed by the user.
- For example, if an app used a service to compact its storage, that would usually be a background service.
- If your app targets API level 26 or higher, the system imposes restrictions on running background services when the app itself isn't in the foreground.
- In most situations, for example, you shouldn't access location information from the background

SERVICES

3. 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, 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.
- Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.

SERVICE LIFE CYCLE



SERVICE LIFE CYCLE

- The lifecycle of a service is much simpler than that of an activity.
- However, it's important because a service can run in the background without the user being aware.
- The service lifecycle—from when it's created to when it's destroyed—can follow either of these two paths:

1. A started service

- The service is created when another component calls startService().
- The service then runs indefinitely and must stop itself by calling stopSelf().
- Another component can also stop the service by calling stopService().
- When the service is stopped, the system destroys it.

SERVICE LIFE CYCLE

2. A bound service

- The service is created when another component (a client) calls bindService().
- The client then communicates with the service through an IBinder interface.
- The client can close the connection by calling unbindService().
- Multiple clients can bind to the same service and when all of them unbind, the system destroys the service.
- The service does not need to stop itself.

BROADCASTS

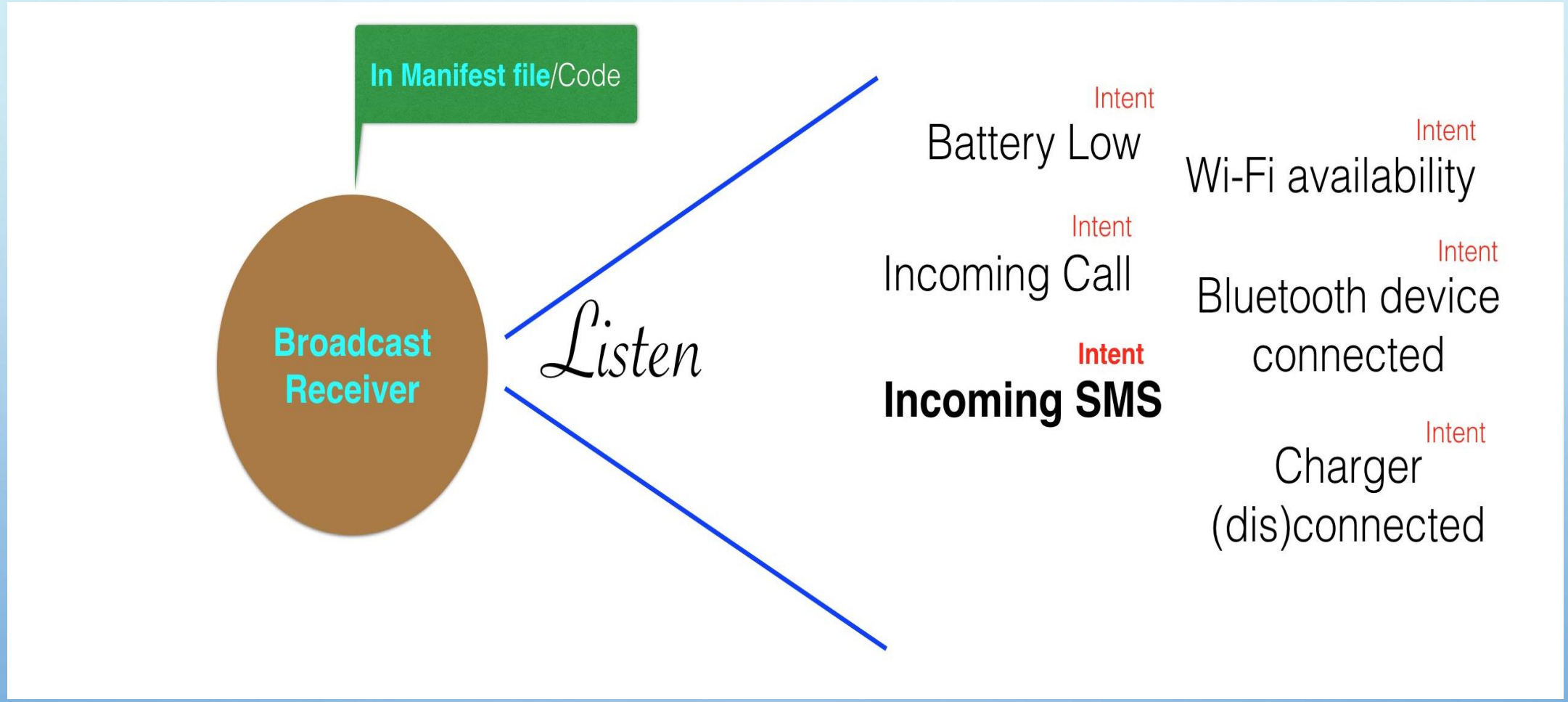
BROADCASTS

- Android apps can send or receive broadcast messages from the Android system and other Android apps, similar to the **publish-subscribe** design pattern.
- These broadcasts are sent when an event of interest occurs.
- For example, the Android system sends broadcasts when various system events occur, such as when the system boots up or the device starts charging.
- Apps can also send custom broadcasts, for example, to notify other apps of something that they might be interested in (for example, some new data has been downloaded).

BROADCAST

- Apps can register to receive specific broadcasts.
- When a broadcast is sent, the system automatically routes broadcasts to apps that have subscribed to receive that particular type of broadcast.
- Generally speaking, broadcasts can be used as a messaging system across apps and outside of the normal user flow.
- However, you must be careful not to abuse the opportunity to respond to broadcasts and run jobs in the background that can contribute to a slow system performance

BROADCAST RECEIVERS



BROADCAST RECEIVERS

- The system automatically sends broadcasts when various system events occur, such as when the system switches in and out of airplane mode.
- System broadcasts are sent to all apps that are subscribed to receive the event.
- The broadcast message itself is wrapped in an Intent object whose action string identifies the event that occurred (for example android.intent.action.AIRPLANE_MODE).
- The intent may also include additional information bundled into its extra field.
- For example, the airplane mode intent includes a boolean extra that indicates whether or not Airplane Mode is on.

BROADCAST RECEIVERS

- Apps can receive broadcasts in two ways:

1. Through manifest-declared receivers **Static Broadcast Receivers:** These types of Receivers are declared in the manifest file and works even if the app is closed.
2. context-registered receivers **Dynamic Broadcast Receivers:** These types of receivers work only if the app is active or minimized.

- If you declare a broadcast receiver in your manifest, the system launches your app (if the app is not already running) when the broadcast is sent.
- If your app targets API level 26 or higher, you cannot use the manifest to declare a receiver for *implicit* broadcasts (broadcasts that do not target your app specifically), except for a few

BROADCAST RECEIVERS

- To declare a broadcast receiver in the manifest, perform the following steps:

1. Specify the <receiver> element in your app's manifest.

```
<receiver android:name=".MyBroadcastReceiver"  android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
        <action android:name="android.intent.action.INPUT_METHOD_CHANGED" />
    </intent-filter>
</receiver>
```

- The intent filters specify the broadcast actions your receiver subscribes to.

BROADCAST RECEIVERS

2. Subclass BroadcastReceiver and implement onReceive(Context, Intent). The broadcast receiver in the following example displays a Toast:

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

BROADCAST RECEIVERS

Context-registered receivers

1. Create an instance of BroadcastReceiver.

```
BroadcastReceiver br = new MyBroadcastReceiver();
```

2. Create an IntentFilter and register the receiver by calling registerReceiver(BroadcastReceiver, IntentFilter):

```
IntentFilter filter = new IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION);
filter.addAction(Intent.ACTION_AIRPLANE_MODE_CHANGED);
this.registerReceiver(br, filter);
```

- Context-registered receivers receive broadcasts as long as their registering context is valid.
- For an example, if you register within an Activity context, you receive broadcasts as long as the activity is not destroyed.

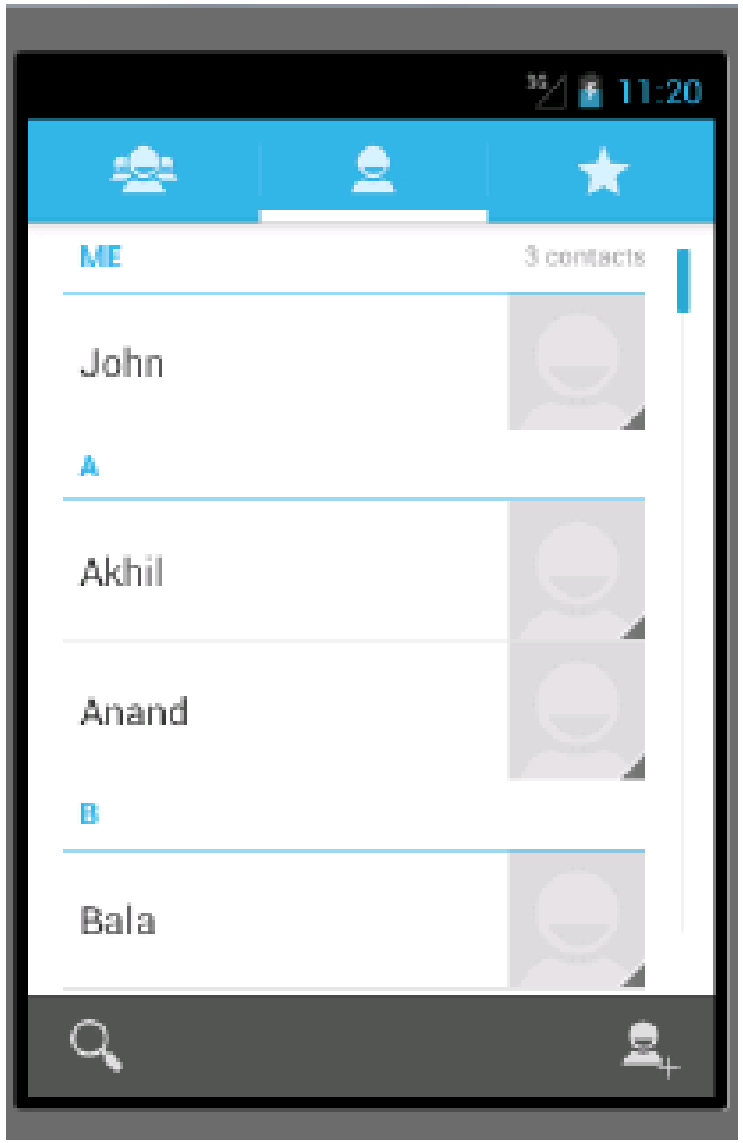
BROADCAST RECEIVERS

Context-registered receivers

3. To stop receiving broadcasts, call `unregisterReceiver(android.content.BroadcastReceiver)`.

- Be sure to unregister the receiver when you no longer need it or the context is no longer valid.
- Be mindful of where you register and unregister the receiver, for example, if you register a receiver in onCreate(Bundle) using the activity's context, you should unregister it in onDestroy().

CONTENT PROVIDERS

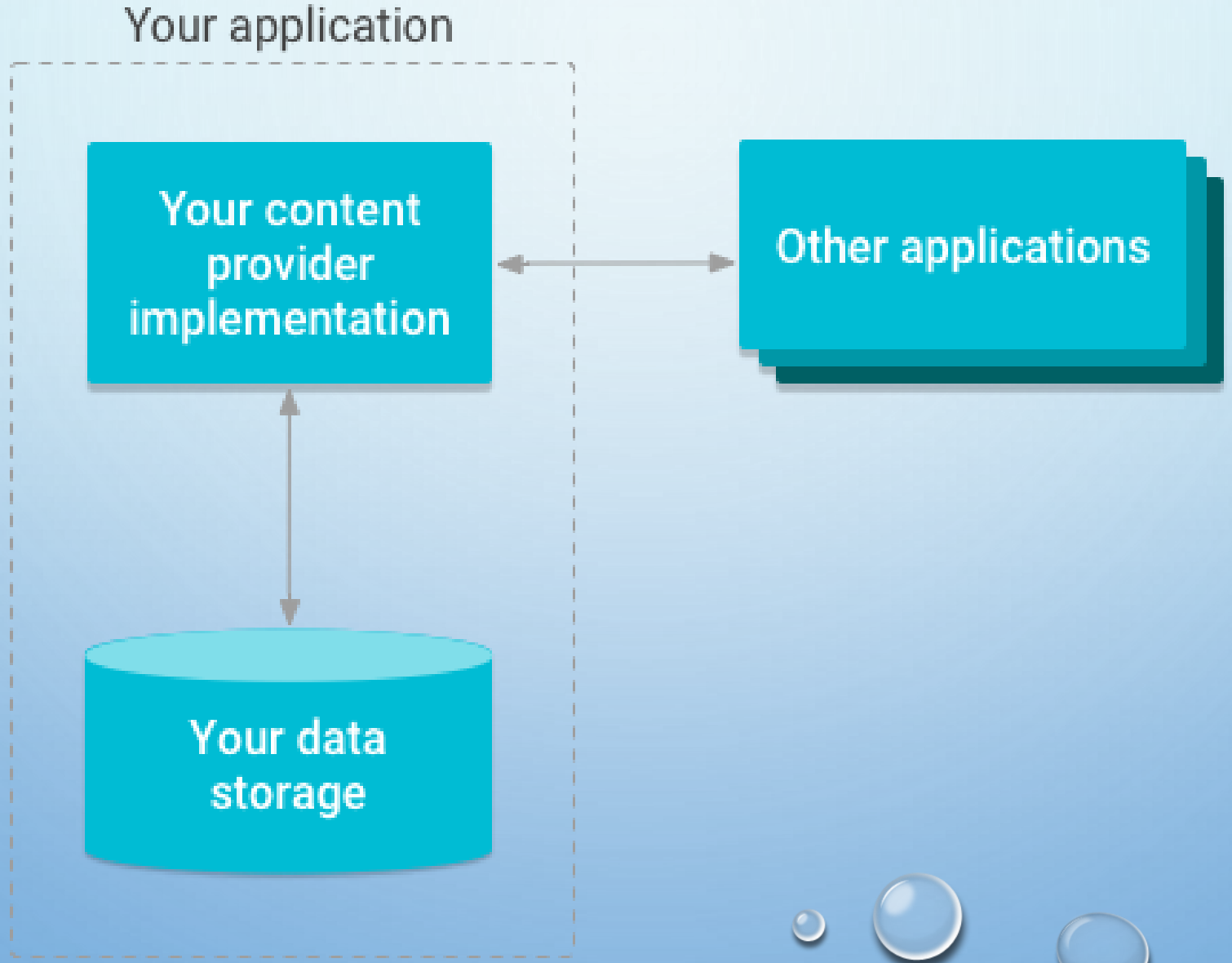


E.g., List of Contacts and the list is shared with the SMS Application

CONTENT PROVIDERS

- Content providers can help an application manage access to data stored by itself, stored by other apps, and provide a way to share data with other apps.
- They encapsulate the data, and provide mechanisms for defining data security.
- Content providers are the standard interface that connects data in one process with code running in another process.
- Implementing a content provider has many advantages.
- Most importantly you can configure a content provider to allow other applications to securely access and modify your app data

CONTENT PROVIDERS



CONTENT PROVIDERS

- A content provider component supplies data from one application to others on request.
- 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.
- 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.

CONTENT PROVIDERS

Content URIs

- To query a content provider, you specify the query string in the form of a URI which has following format −

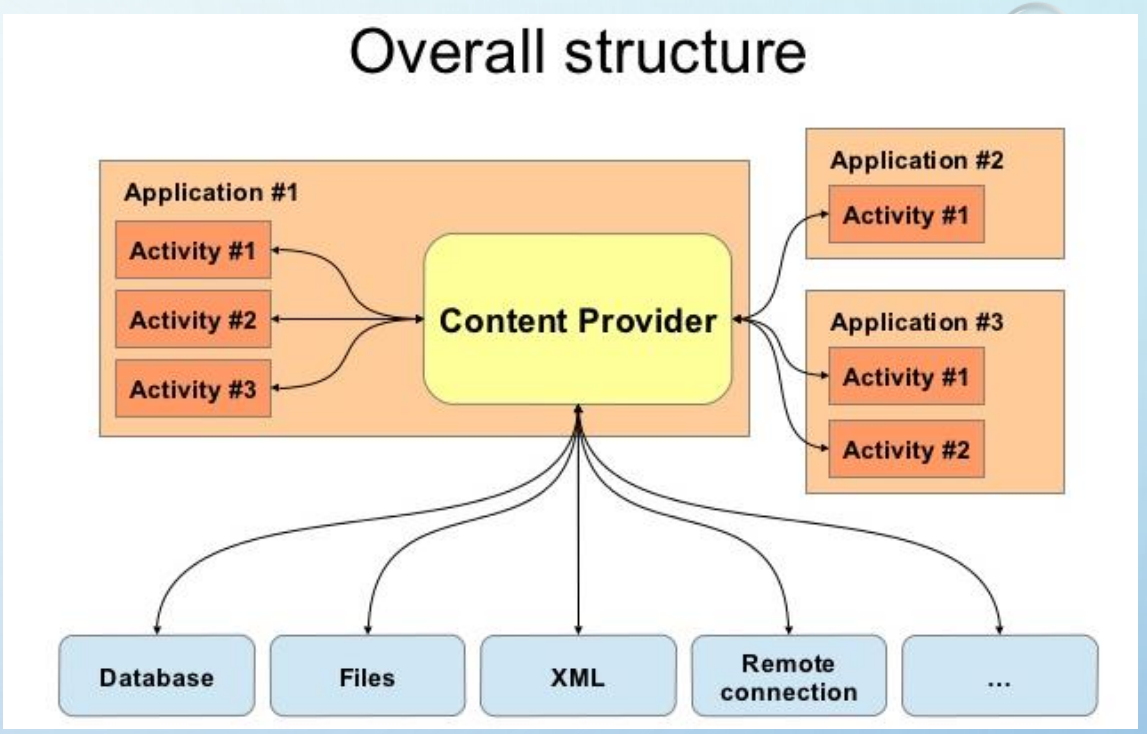
```
<prefix>://<authority>/<data_type>/<id>
```

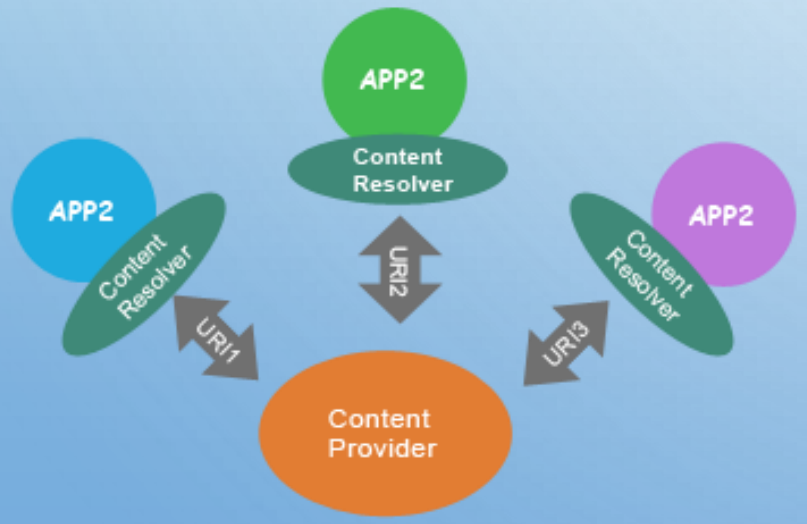
```
E.g. content://edu.aimit.MyApp.StudentsProvider
```

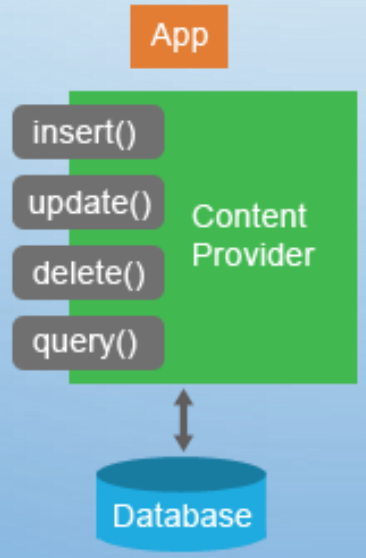
CONTENT PROVIDERS

- A list of methods which you need to override in Content Provider class to have your Content Provider working
- **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()** inserts a new record into the content provider.
- **delete()** deletes an existing record from the content provider.
- **update()** updates an existing record from the content provider.
- **getType()** This returns the MIME type of the data at the given URI.

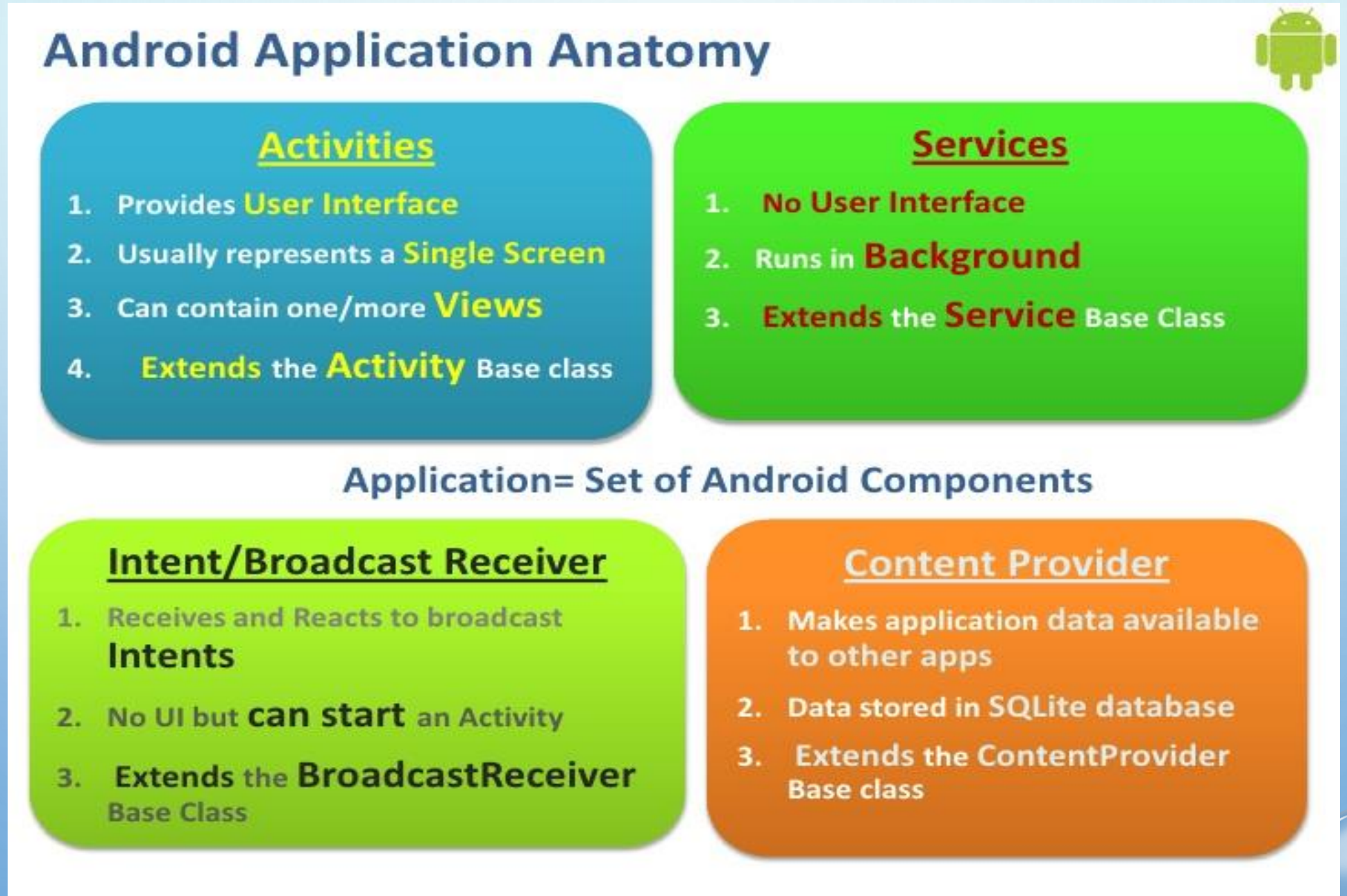
CONTENT PROVIDERS







RECAP


Android Application Anatomy
Activities
1. Provides User Interface
2. Usually represents a Single Screen
3. Can contain one/more Views
4. Extends the Activity Base class
Services
1. No User Interface
2. Runs in Background
3. Extends the Service Base Class
Application= Set of Android Components
Intent/Broadcast Receiver
1. Receives and Reacts to broadcast Intents
2. No UI but can start an Activity
3. Extends the BroadcastReceiver Base Class
Content Provider
1. Makes application data available to other apps
2. Data stored in SQLite database
3. Extends the ContentProvider Base class