UNIT III

Activities Life Cycle and Working

Broadcast Receivers

Content Providers

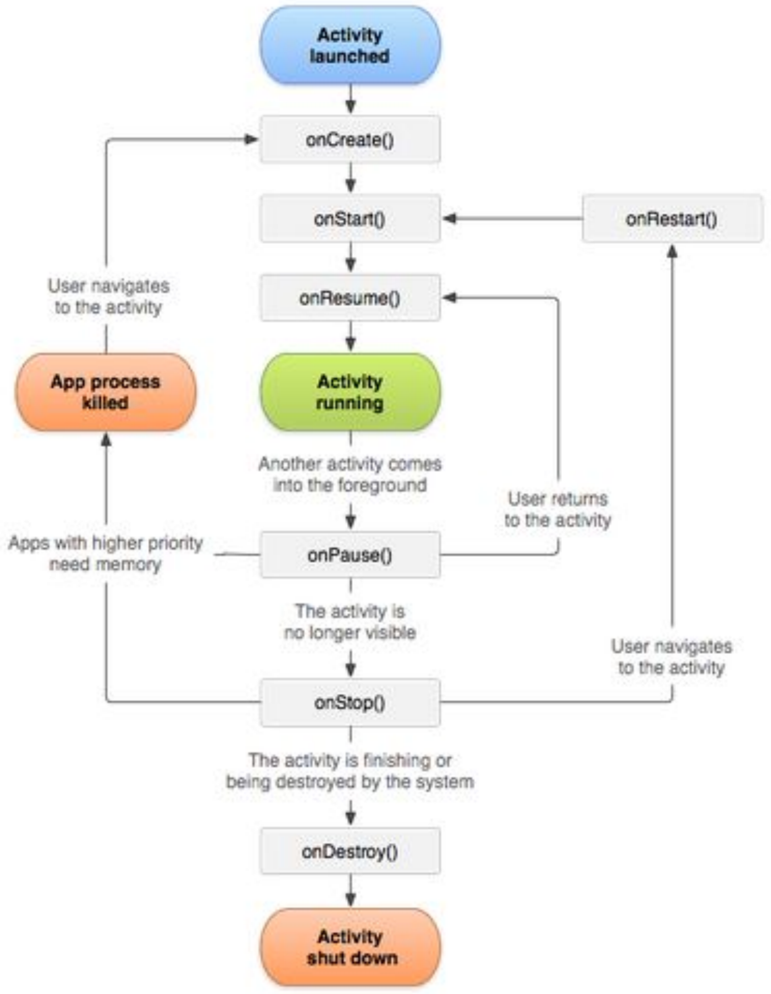
Services

- As a user navigates through, out of, and back to your app, the Activity instances in your app transition through different states in their lifecycle.
- The Activity class provides a number of callbacks that allow the activity to know that a state has changed: that the system is creating, stopping, or resuming an activity, or destroying the process in which the activity resides.
- Within the lifecycle callback methods, you can declare how your activity behaves when the user leaves and re-enters the activity.
    - For example, if you're building a streaming video player, you might pause the video and terminate the network connection when the user switches to another app.
    - When the user returns, you can reconnect to the network and allow the user to resume the video from the same spot.
- Doing the right work at the right time and handling transitions properly make your app more robust and performant.

- Good implementation of the lifecycle callbacks can help ensure that your app avoids:
  - Crashing if the user receives a phone call or switches to another app while using your app.
  - Consuming valuable system resources when the user is not actively using it.
  - Losing the user's progress if they leave your app and return to it at a later time.
  - Crashing or losing the user's progress when the screen rotates between landscape and portrait orientation.

• Activity-lifecycle concepts

- To navigate transitions between stages of the activity lifecycle, the Activity class provides a core set of six callbacks: onCreate(), onStart(), onResume(), onPause(), onStop(), and onDestroy().
- The system invokes each of these callbacks as an activity enters a new state.


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

- As the user begins to leave the activity, the system calls methods to dismantle the activity. In some cases, this dismantlement is only partial; the activity still resides in memory (such as when the user switches to another app), and can still come back to the foreground.

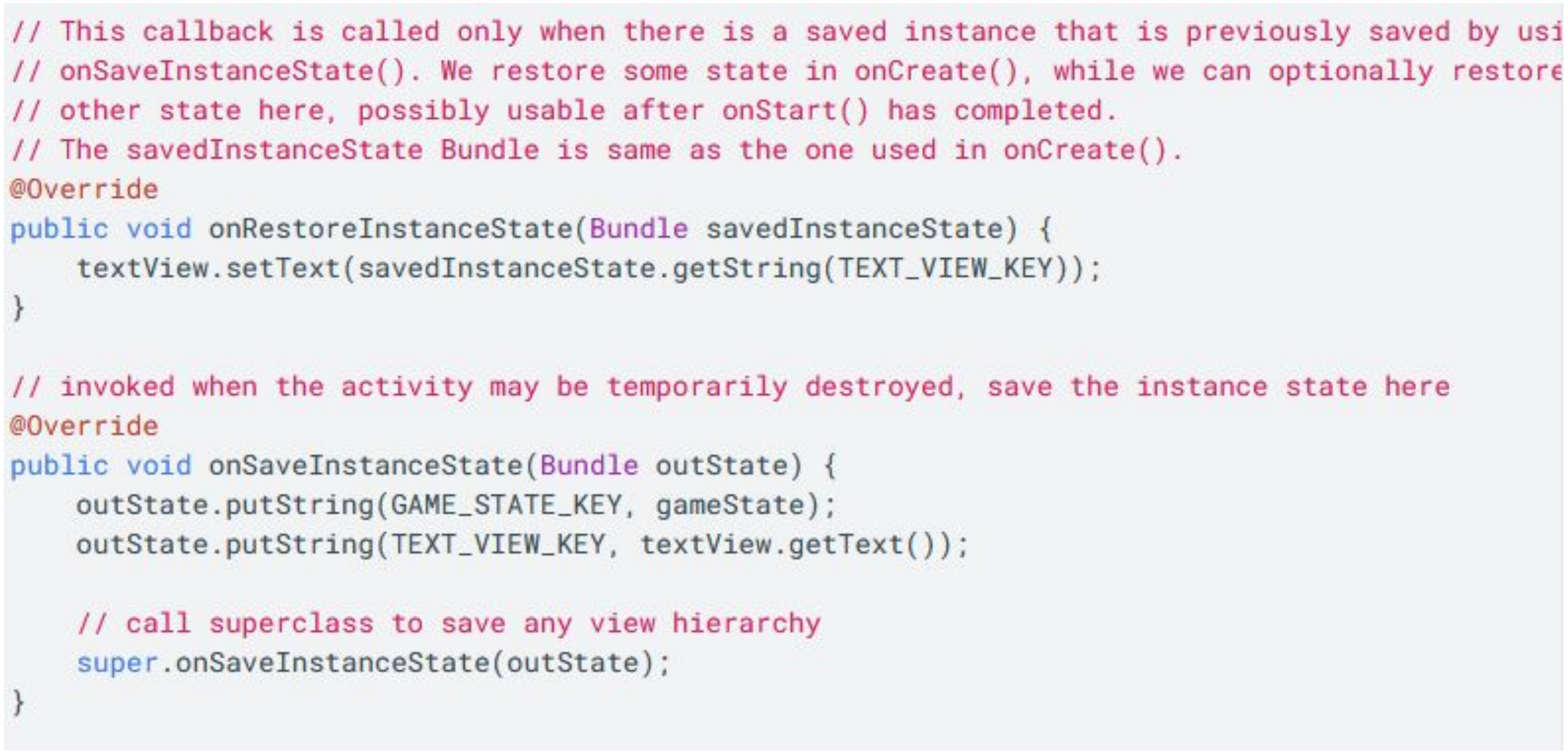
- If the user returns to that activity, the activity resumes from where the user left off. With a few exceptions, apps are restricted from starting activities when running in the background.

- The system's likelihood of killing a given process—along with the activities in it—depends on the state of the activity at the time. Activity state and ejection from memory provides more information on the relationship between state and vulnerability to ejection.

- **Lifecycle callbacks**
  - **onCreate()**
    - 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.
      - For example, your implementation of onCreate() might bind data to lists, associate the activity with a ViewModel, and instantiate some class-scope variables.
    - This method receives the parameter savedInstanceState, which is a Bundle object containing the activity's previously saved state. If the activity has never existed before, the value of the Bundle object is null.

    - If you have a lifecycle-aware component that is hooked up to the lifecycle of your activity it will receive the ON_CREATE event.
    - The method annotated with @OnLifecycleEvent will be called so your lifecycle-aware component can perform any setup code it needs for the created state.

```
TextView textView;

// some transient state for the activity instance
String gameState;

@Override
public void onCreate(Bundle savedInstanceState) {
    // call the super class onCreate to complete the creation of activity like
    // the view hierarchy
    super.onCreate(savedInstanceState);

    // recovering the instance state
    if (savedInstanceState != null) {
        gameState = savedInstanceState.getString(GAME_STATE_KEY);
    }

    // set the user interface layout for this activity
    // the layout file is defined in the project res/layout/main_activity.xml file
    setContentView(R.layout.main_activity);

    // initialize member TextView so we can manipulate it later
    textView = (TextView) findViewById(R.id.text_view);
}
```

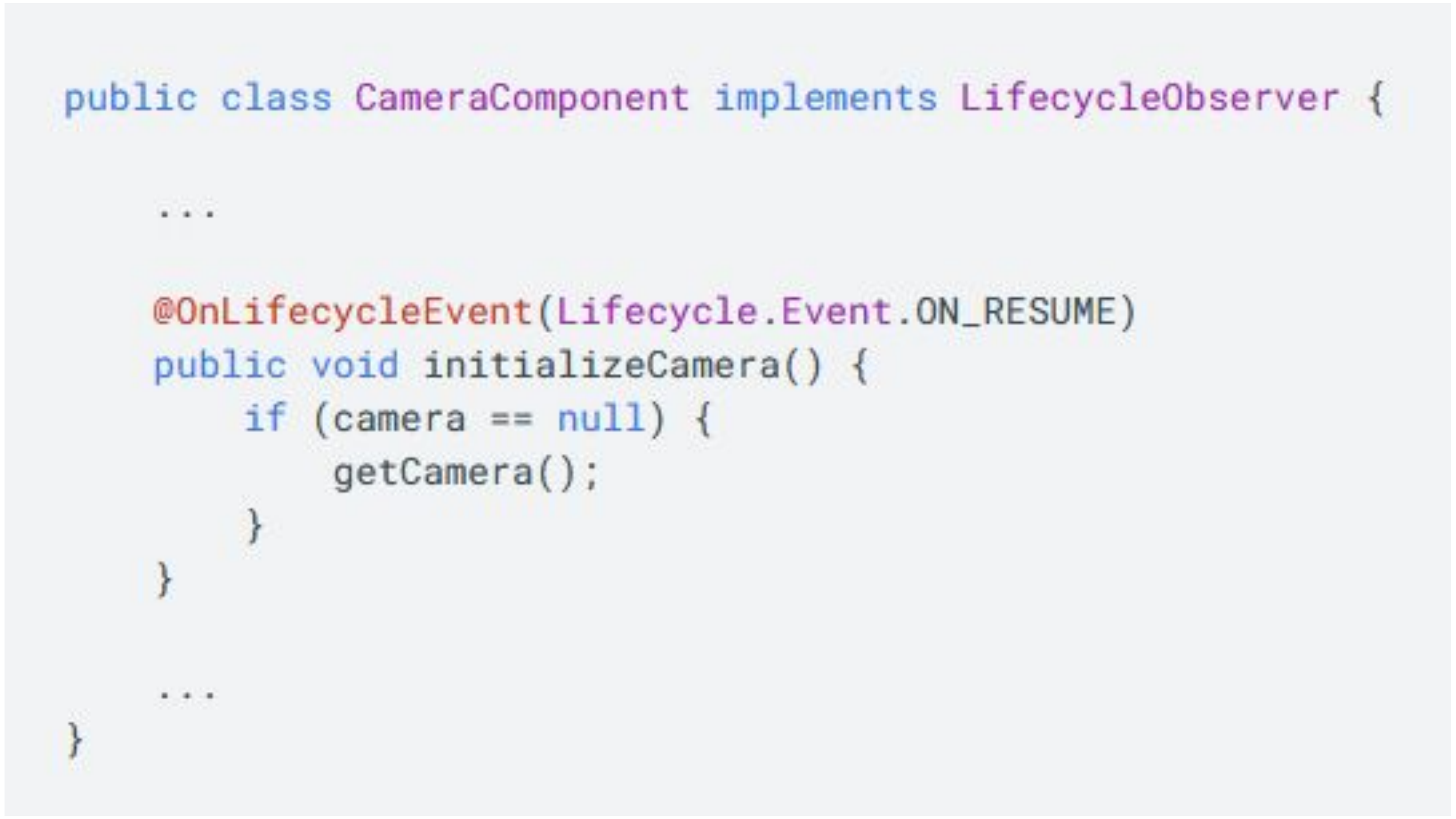


```java
// This callback is called only when there is a saved instance that is previously saved by usi
// onSaveInstanceState(). We restore some state in onCreate(), while we can optionally restore
// other state here, possibly usable after onStart() has completed.
// The savedInstanceState Bundle is same as the one used in onCreate().
@Override
public void onRestoreInstanceState(Bundle savedInstanceState) {
    textView.setText(savedInstanceState.getString(TEXT_VIEW_KEY));
}

// invoked when the activity may be temporarily destroyed, save the instance state here
@Override
public void onSaveInstanceState(Bundle outState) {
    outState.putString(GAME_STATE_KEY, gameState);
    outState.putString(TEXT_VIEW_KEY, textView.getText());

    // call superclass to save any view hierarchy
    super.onSaveInstanceState(outState);
}
```

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.

- **onStart()**
  - 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.

- **onResume()**
  - 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.
  - When the activity moves to the resumed state, any lifecycle-aware component tied to the activity's lifecycle will receive the ON_RESUME event. This is where the lifecycle components can enable any functionality that needs to run while the component is visible and in the foreground, such as starting a camera preview.
  - When an interruptive event occurs, the activity enters the Paused state, and the system invokes the onPause() callback.
  - If the activity returns to the Resumed state from the Paused state, the system once again calls onResume() method. For this reason, you should implement onResume() to initialize components that you release during onPause(), and perform any other initializations that must occur each time the activity enters the Resumed state.

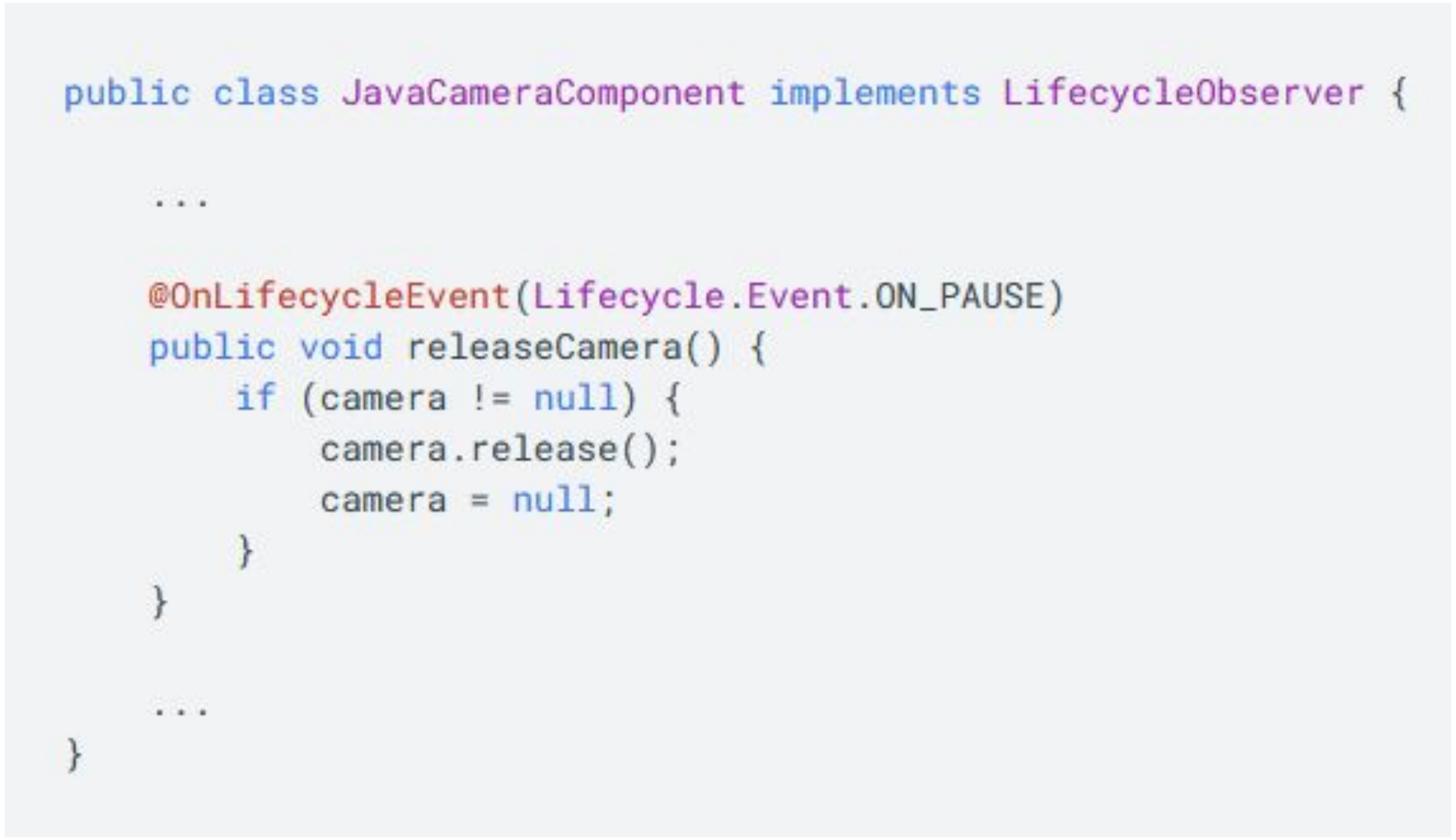


```java
public class CameraComponent implements LifecycleObserver {

    ...

    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
    public void initializeCamera() {
        if (camera == null) {
            getCamera();
        }
    }

    ...
}
```

The code above initializes the camera once the LifecycleObserver receives the ON_RESUME event.

- **onPause()**
  - 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).
  - Use the onPause() method to pause or adjust operations that should not continue (or should continue in moderation) while the Activity is in the Paused state, and that you expect to resume shortly.
  - 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.
    -

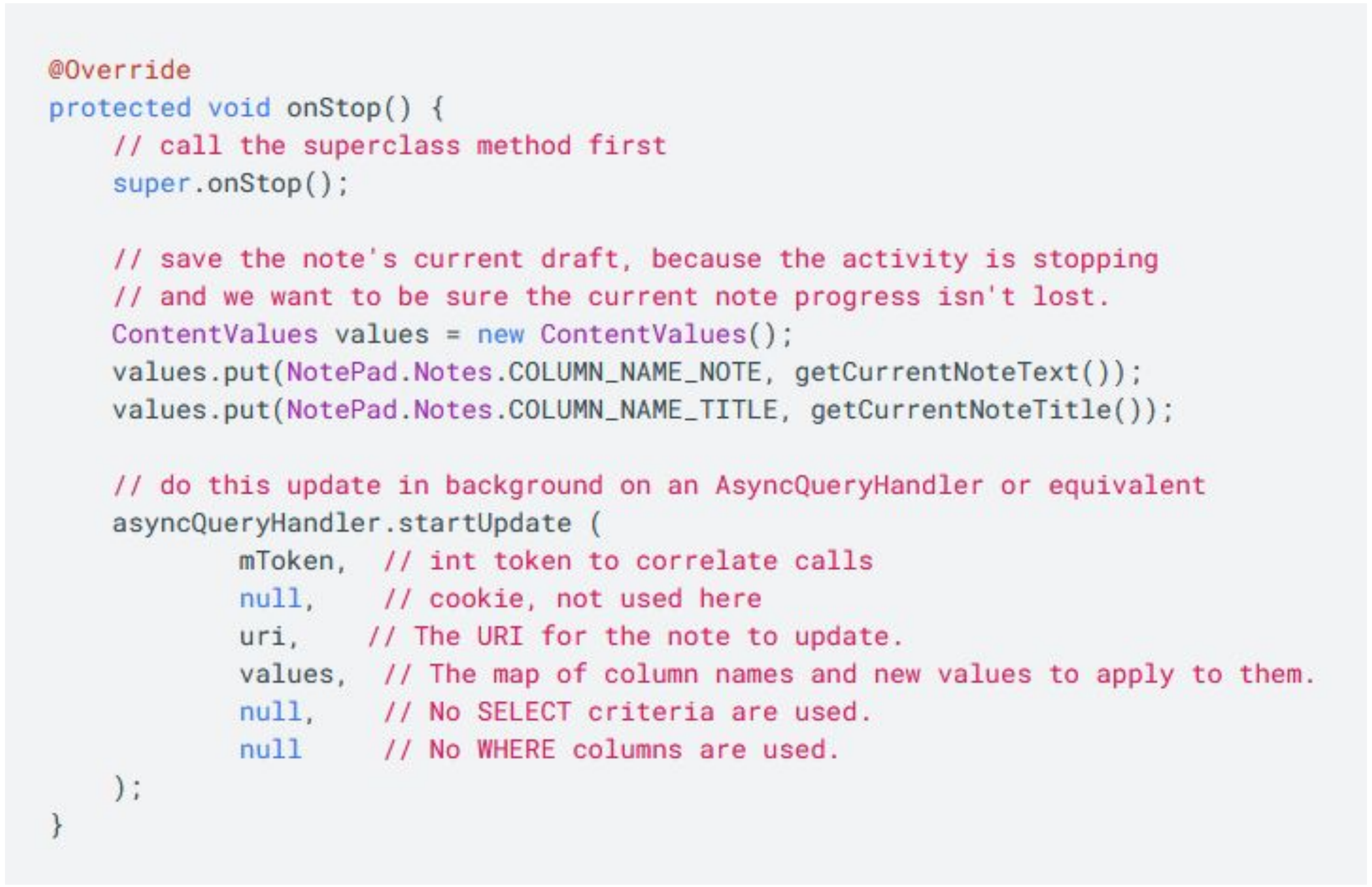
- When the activity moves to the paused state, any lifecycle-aware component tied to the activity's lifecycle will receive the ON_PAUSE event. This is where the lifecycle components can stop any functionality that does not need to run while the component is not in the foreground, such as stopping a camera preview.



```
public class JavaCameraComponent implements LifecycleObserver {

    ...

    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
    public void releaseCamera() {
        if (camera != null) {
            camera.release();
            camera = null;
        }
    }

    ...
}
```

- **onStop()**
  - 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.
  - When the activity moves to the stopped state, any lifecycle-aware component tied to the activity's lifecycle will receive the ON_STOP event. This is where the lifecycle components can stop any functionality that does not need to run while the component is not visible on the screen
  - 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, your app might pause animations or switch from fine-grained to coarse-grained location updates. Using onStop() instead of onPause() ensures that UI-related work continues, even when the user is viewing your activity in multi-window mode.

- You should also use onStop() to perform relatively CPU-intensive shutdown operations. For example, if you can't find a more opportune time to save information to a database, you might do so during onStop().



```java
@Override
protected void onStop() {
    // call the superclass method first
    super.onStop();

    // save the note's current draft, because the activity is stopping
    // and we want to be sure the current note progress isn't lost.
    ContentValues values = new ContentValues();
    values.put(NotePad.Notes.COLUMN_NAME_NOTE, getCurrentNoteText());
    values.put(NotePad.Notes.COLUMN_NAME_TITLE, getCurrentNoteTitle());

    // do this update in background on an AsyncQueryHandler or equivalent
    asyncQueryHandler.startUpdate (
            mToken,  // int token to correlate calls
            null,    // cookie, not used here
            uri,    // The URI for the note to update.
            values,  // The map of column names and new values to apply to them.
            null,    // No SELECT criteria are used.
            null     // No WHERE columns are used.
    );
}
```

- When your activity enters the Stopped state, the Activity object is kept resident in memory: It maintains all state and member information, but is not attached to the window manager.
- When the activity resumes, the activity recalls this information.

- **onDestroy()**
  - onDestroy() 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)
  - When the activity moves to the destroyed state, any lifecycle-aware component tied to the activity's lifecycle will receive the ON_DESTROY event. This is where the lifecycle components can clean up anything it needs to before the Activity is destroyed.
  - The onDestroy() callback should release all resources that have not yet been released by earlier callbacks such as onStop().

Views, Layouts

- The basic building block for user interface is a view
- View is created from the View class and occupies a rectangular area on the screen
- View is responsible for drawing and event handling
- View is the base class for widgets, which are used to create interactive UI components like button, textfiles etc

View group

- The View group is a subclass of View and provides invisible container that hold other View or other View Group and define their layout property
- Linear layout is a View Group
- View is a single element

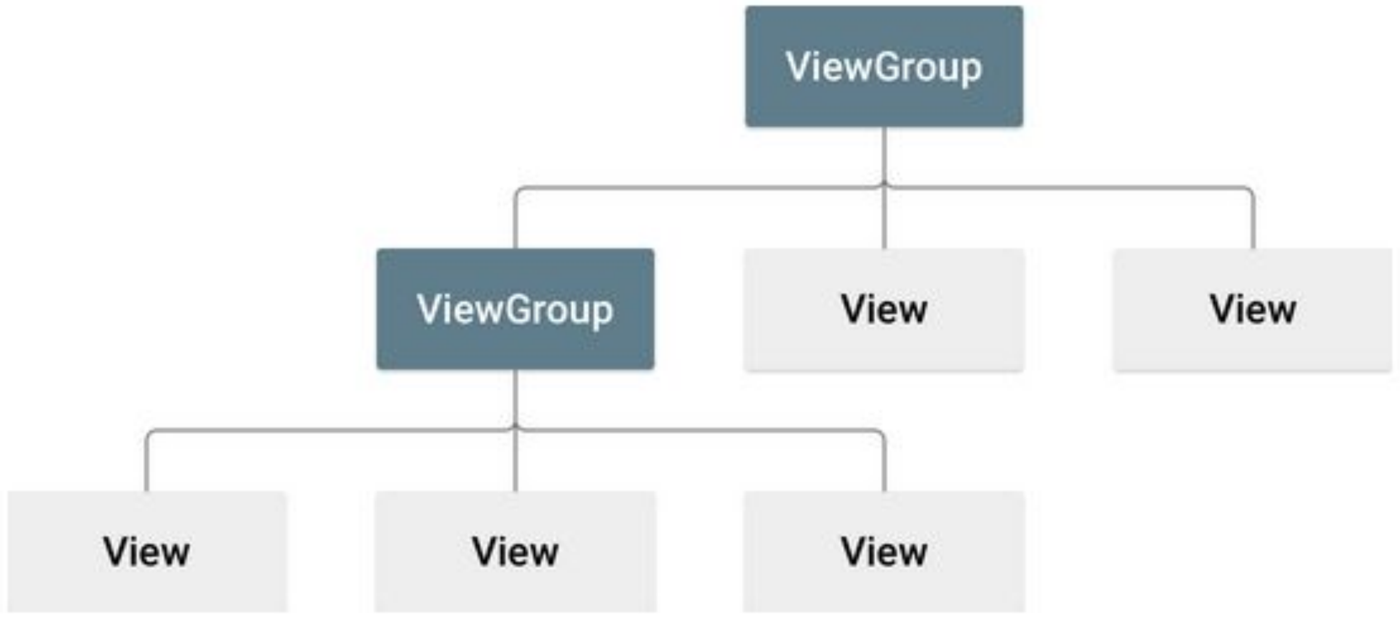
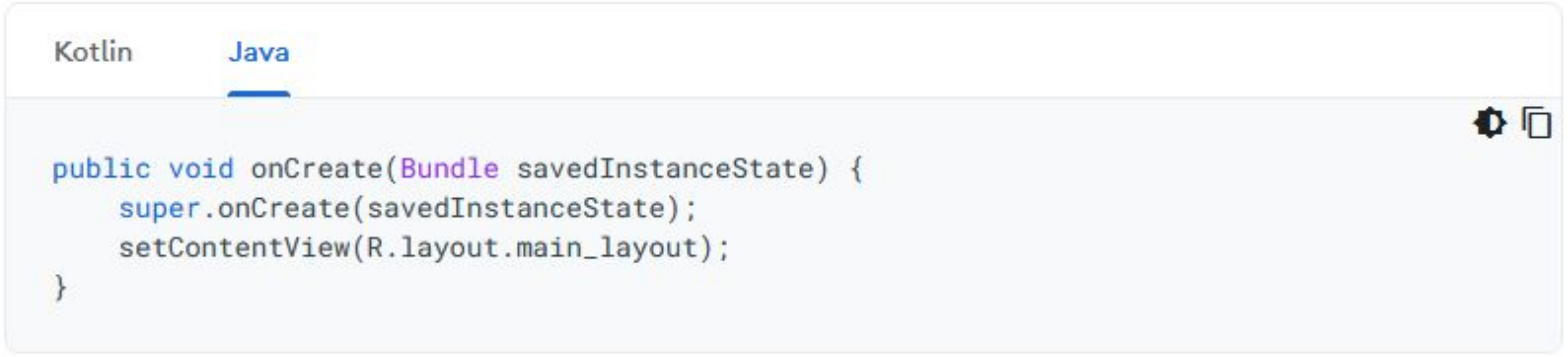


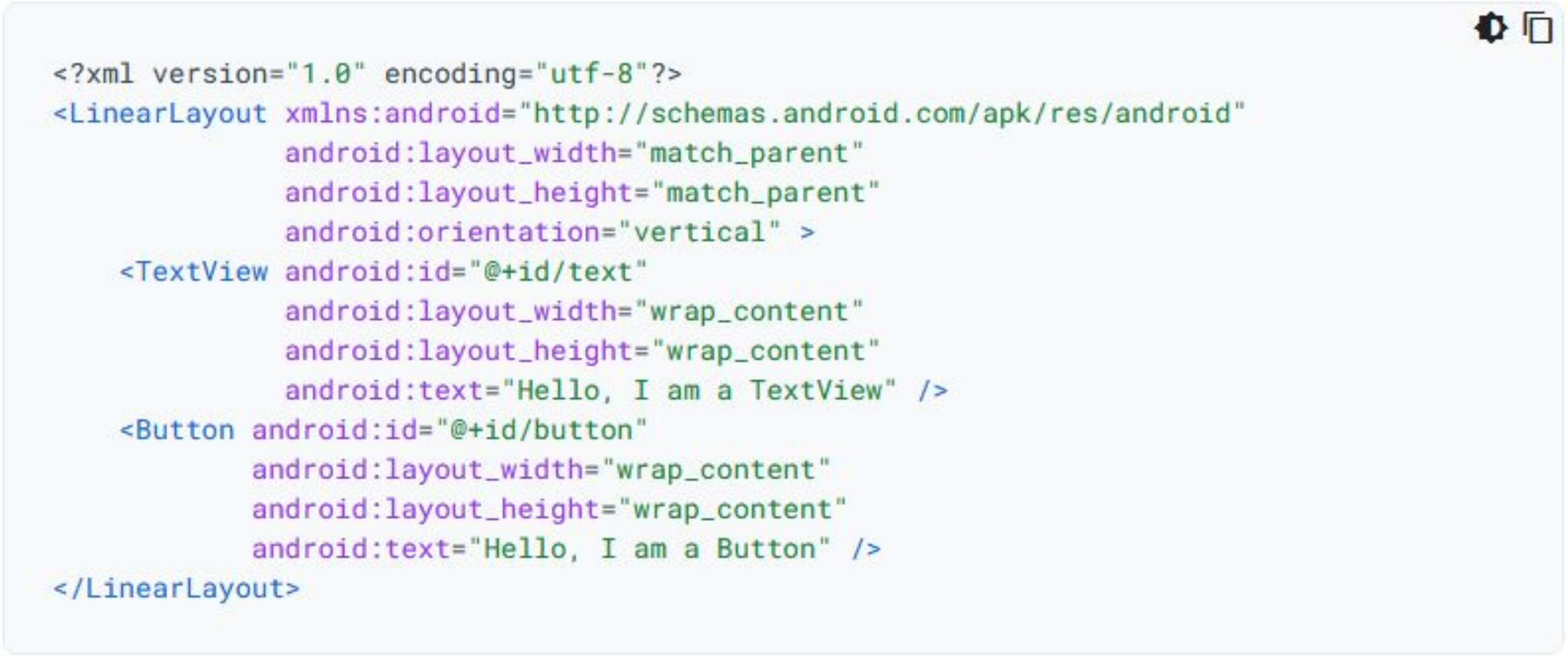
Figure 1. Illustration of a view hierarchy, which defines a UI layout

- Using Android's XML vocabulary, you can quickly design UI layouts and the screen elements they contain, in the same way you create web pages in HTML — with a series of nested elements.
- Each layout file must contain exactly one root element, which must be a View or ViewGroup object. Once you've defined the root element, you can add additional layout objects or widgets as child elements to gradually build a View hierarchy that defines your layout.

- **Load the XML Resource**
  - When you compile your app, each XML layout file is compiled into a View resource. You should load the layout resource from your app code, in your Activity.onCreate() callback implementation.
  - Do so by calling setContentView(), passing it the reference to your layout resource in the form of:
    - R.layout.layout_file_name
  - The onCreate() callback method in your Activity is called by the Android framework when your Activity is launched



```java
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main_layout);
}
```



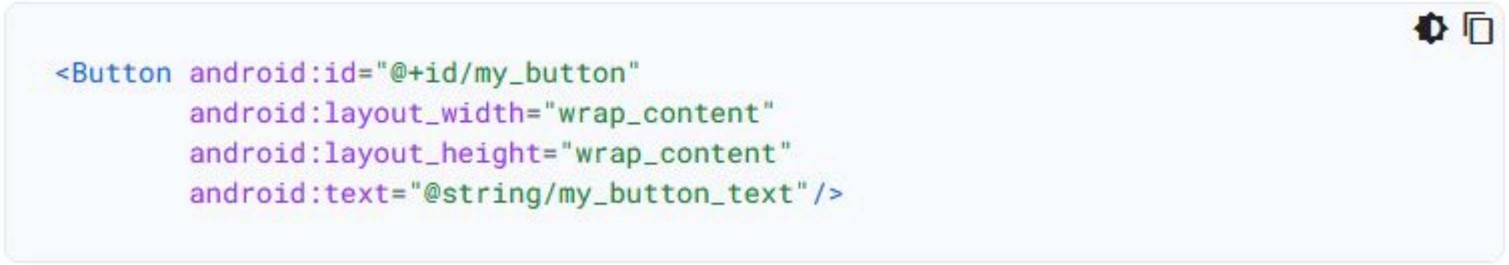
```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical" >
    <TextView android:id="@+id/text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello, I am a Button" />
</LinearLayout>
```

- **Attributes**
  - Every View and ViewGroup object supports their own variety of XML attributes.

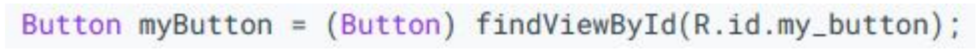
  - Some attributes are specific to a View object (for example, TextView supports the textSize attribute), but these attributes are also inherited by any View objects that may extend this class.
  - Some are common to all View objects, because they are inherited from the root View class (like the id attribute).
  - And, other attributes are considered "layout parameters," which are attributes that describe certain layout orientations of the View object, as defined by that object's parent ViewGroup object.

• ID

- Any View object may have an integer ID associated with it, to uniquely identify the View within the tree. When the app is compiled, this ID is referenced as an integer, but the ID is typically assigned in the layout XML file as a string, in the id attribute.
- This is an XML attribute common to all View objects (defined by the View class)
  - android:id="@+id/my_button“
- The at-symbol (@) at the beginning of the string indicates that the XML parser should parse and expand the rest of the ID string and identify it as an ID resource.
- The plus-symbol (+) means that this is a new resource name that must be created and added to our resources (in the R.java file).
- When referencing an Android resource ID, you do not need the plus-symbol, but must add the android package namespace, like so:
  - android:id="@android:id/empty"
- With the android package namespace in place, we're now referencing an ID from the android.R resources class, rather than the local resources class.



```
<Button android:id="@+id/my_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/my_button_text"/>
```



```
Button myButton = (Button) findViewById(R.id.my_button);
```

Layout Parameters

- XML layout attributes named layout_something define layout parameters for the View that are appropriate for the ViewGroup in which it resides.
- Every ViewGroup class implements a nested class that extends ViewGroup.LayoutParams.
- This subclass contains property types that define the size and position for each child view, as appropriate for the view group.

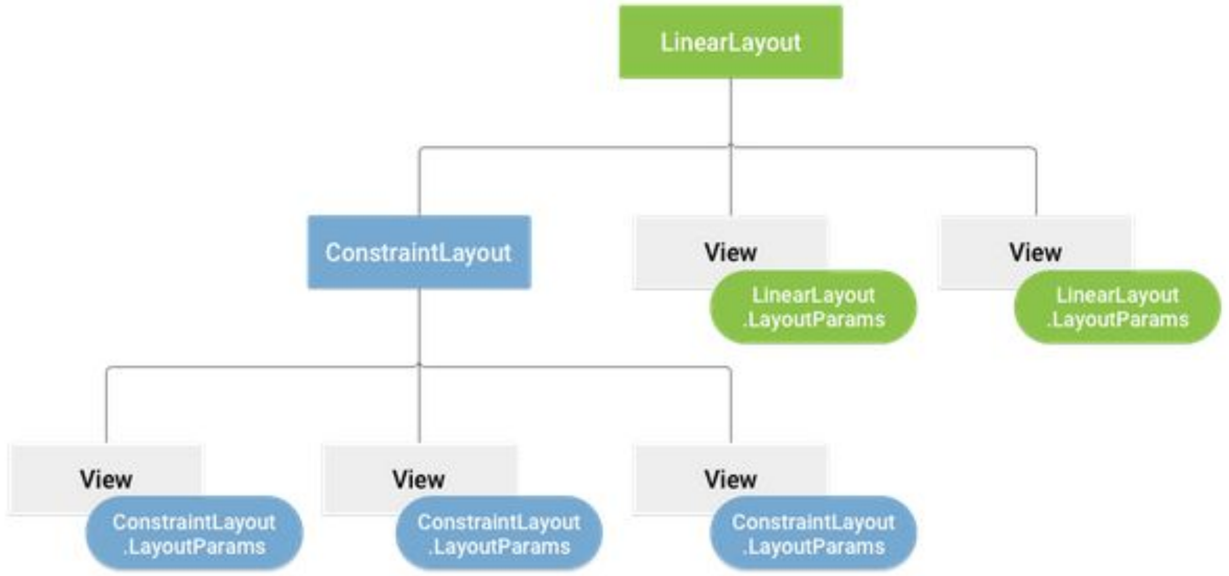

LinearLayout
ConstraintLayout
View
LinearLayout .LayoutParams
View
LinearLayout .LayoutParams
View
ConstraintLayout .LayoutParams
View
ConstraintLayout .LayoutParams
View
ConstraintLayout .LayoutParams

Figure 2. Visualization of a view hierarchy with layout parameters associated with each view

- Note that every LayoutParams subclass has its own syntax for setting values. Each child element must define LayoutParams that are appropriate for its parent, though it may also define different LayoutParams for its own children.

- All view groups include a width and height (layout_width and layout_height), and each view is required to define them.
- You can specify width and height with exact measurements, though you probably won't want to do this often. More often, you will use one of these constants to set the width or height:

  - wrap_content tells your view to size itself to the dimensions required by its content.
  - match_parent tells your view to become as big as its parent view group will allow.

- **Layout Position**
  - The geometry of a view is that of a rectangle. A view has a location, expressed as a pair of left and top coordinates, and two dimensions, expressed as a width and a height. The unit for location and dimensions is the pixel.

  - It is possible to retrieve the location of a view by invoking the methods getLeft() and getTop().
  - The former returns the left, or X, coordinate of the rectangle representing the view. The latter returns the top, or Y, coordinate of the rectangle representing the view.

  - These methods both return the location of the view relative to its parent.

  - getRight() and getBottom(). These methods return the coordinates of the right and bottom edges of the rectangle representing the view.

Layouts

- Layouts are subclasses of ViewGroup class and a typical layout defines the visual structure for an android user interface
- The layout defines the visual structure of your application
- Android studio uses this structure to display the view elements on the screen.
- Layouts are also called view containers
- A layout may contain any type of views such as buttons, labels, textboxes and so on…

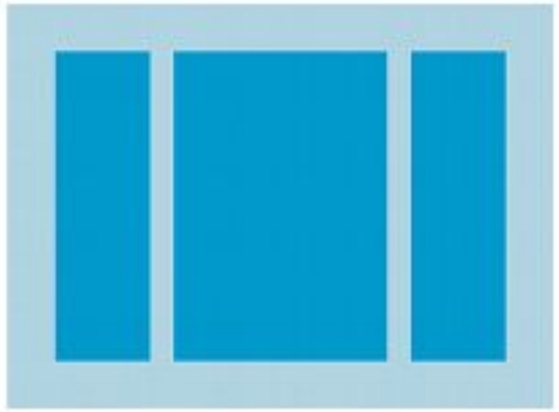
Android studio layout types

- Linear Layout
- Relative layout
- Table layout – Calculator
- Frame Layout
- Grid Layout – Gallery app/Hotel Menu
- Constraint Layout
- Tab layout – what’s app

- Size, Padding and Margins
  - The size of a view is expressed with a width and a height. A view actually possesses two pairs of width and height values.
  - The first pair is known as measured width and measured height. These dimensions define how big a view wants to be within its parent. The measured dimensions can be obtained by calling getMeasuredWidth() and getMeasuredHeight().
  - The second pair is simply known as width and height, or sometimes drawing width and drawing height. These dimensions define the actual size of the view on screen, at drawing time and after layout. These values may, but do not have to, be different from the measured width and height. The width and height can be obtained by calling getWidth() and getHeight().
  - To measure its dimensions, a view takes into account its padding. The padding is expressed in pixels for the left, top, right and bottom parts of the view. Padding can be used to offset the content of the view by a specific number of pixels. For instance, a left padding of 2 will push the view's content by 2 pixels to the right of the left edge. Padding can be set using the setPadding(int, int, int, int) method and queried by calling getPaddingLeft(), getPaddingTop(), getPaddingRight() and getPaddingBottom().

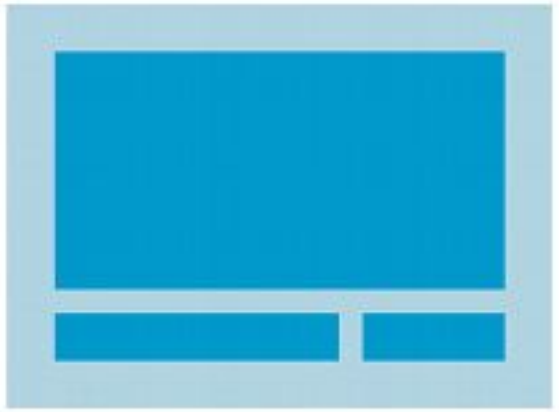
- **Common Layouts**
  - Each subclass of the ViewGroup class provides a unique way to display the views you nest within it.

Linear Layout



A layout that organizes its children into a single horizontal or vertical row. It creates a scrollbar if the length of the window exceeds the length of the screen.

Relative Layout



Enables you to specify the location of child objects relative to each other (child A to the left of child B) or to the parent (aligned to the top of the parent).

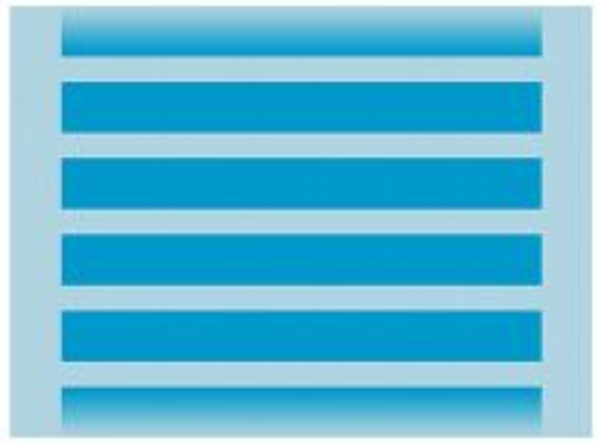
Web View

Displays web pages.

Building Layouts with an Adapter

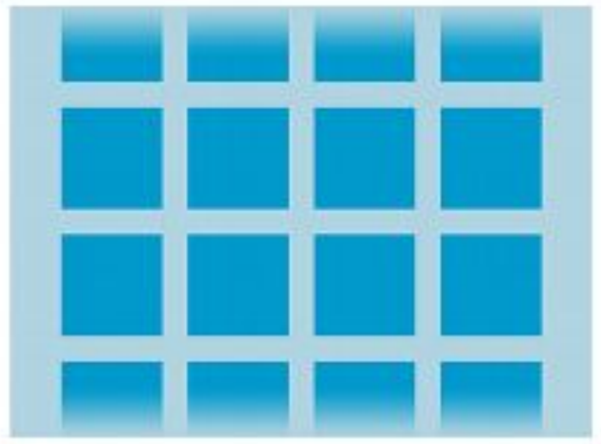
- When the content for your layout is dynamic or not pre-determined, you can use a layout that subclasses AdapterView to populate the layout with views at runtime.
- A subclass of the AdapterView class uses an Adapter to bind data to its layout.
- The Adapter behaves as a middleman between the data source and the AdapterView layout—the Adapter retrieves the data (from a source such as an array or a database query) and converts each entry into a view that can be added into the AdapterView layout.

List View



Displays a scrolling single column list.

Grid View



Displays a scrolling grid of columns and rows.

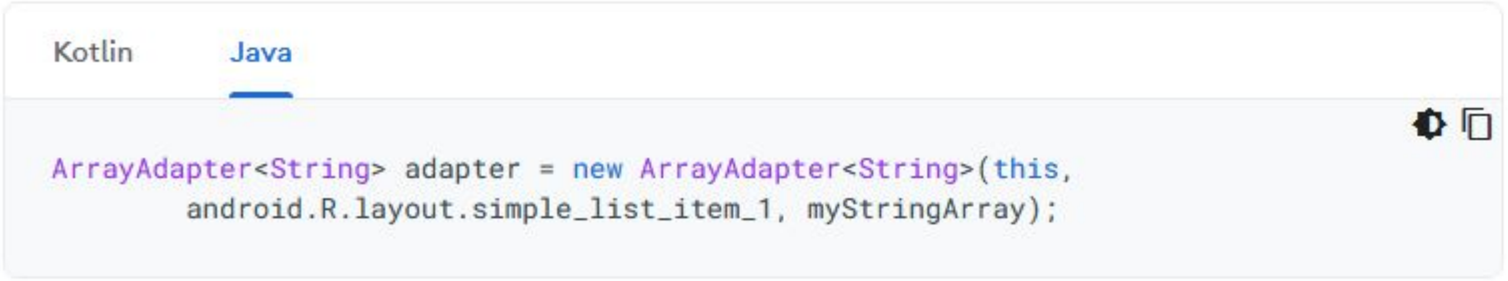
• Filling an adapter view with data

- You can populate an AdapterView such as ListView or GridView by binding the AdapterView instance to an Adapter, which retrieves data from an external source and creates a View that represents each data entry.

- Android provides several subclasses of Adapter that are useful for retrieving different kinds of data and building views for an AdapterView. The two most common adapters are:

- ArrayAdapter
  - Use this adapter when your data source is an array. By default, ArrayAdapter creates a view for each array item by calling toString() on each item and placing the contents in a TextView.

  - For example, if you have an array of strings you want to display in a ListView, initialize a new ArrayAdapter using a constructor to specify the layout for each string and the string array:

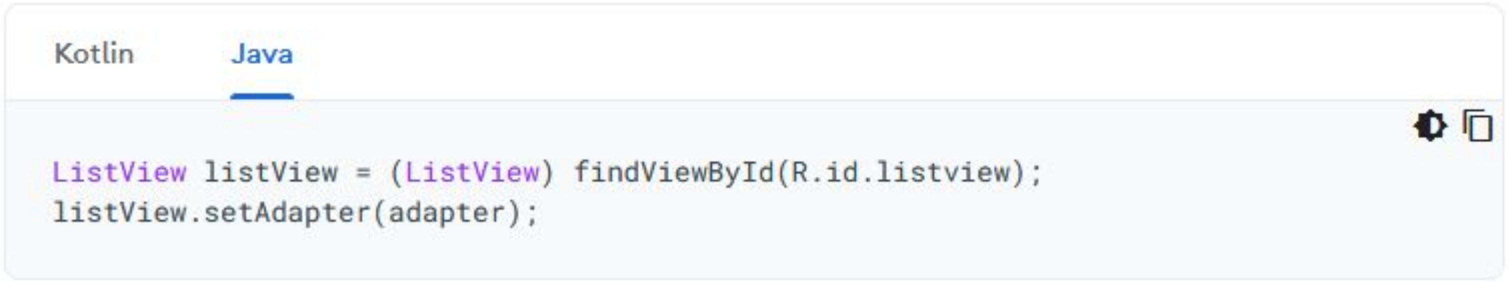


```java
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
        android.R.layout.simple_list_item_1, myStringArray);
```

- The arguments for this constructor are:

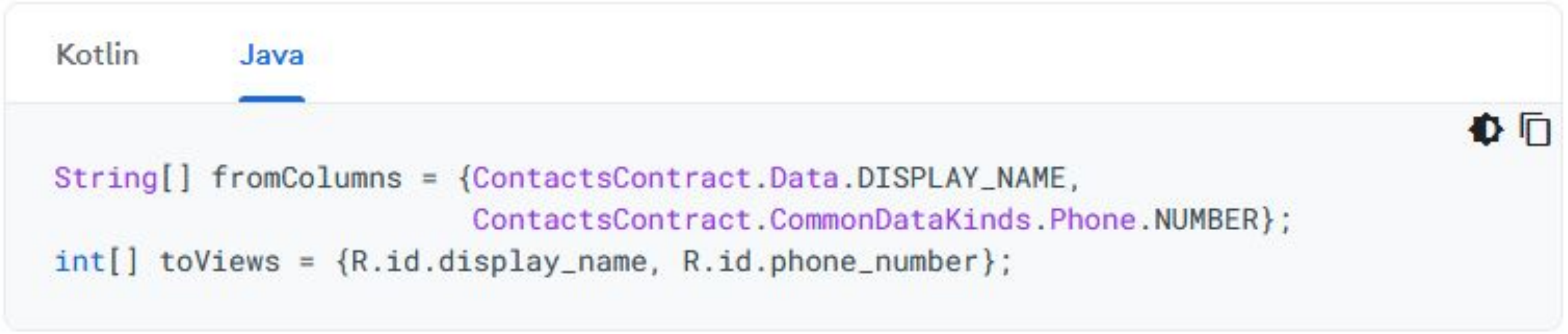
  - Your app Context
  - The layout that contains a TextView for each string in the array
  - The string array

- Then simply call setAdapter() on your ListView:



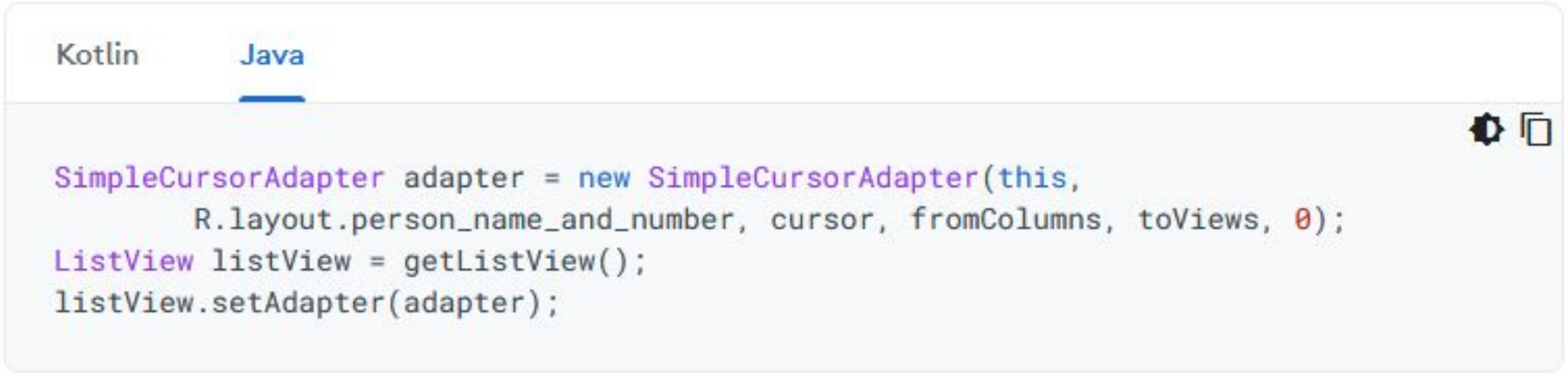
```
ListView listView = (ListView) findViewById(R.id.listview);
listView.setAdapter(adapter);
```

- SimpleCursorAdapter
  - Use this adapter when your data comes from a Cursor. When using SimpleCursorAdapter, you must specify a layout to use for each row in the Cursor and which columns in the Cursor should be inserted into which views of the layout.
  - For example, if you want to create a list of people's names and phone numbers, you can perform a query that returns a Cursor containing a row for each person and columns for the names and numbers.
  - You then create a string array specifying which columns from the Cursor you want in the layout for each result and an integer array specifying the corresponding views that each column should be placed:



```java
String[] fromColumns = {ContactsContract.Data.DISPLAY_NAME,
                        ContactsContract.CommonDataKinds.Phone.NUMBER};
int[] toViews = {R.id.display_name, R.id.phone_number};
```

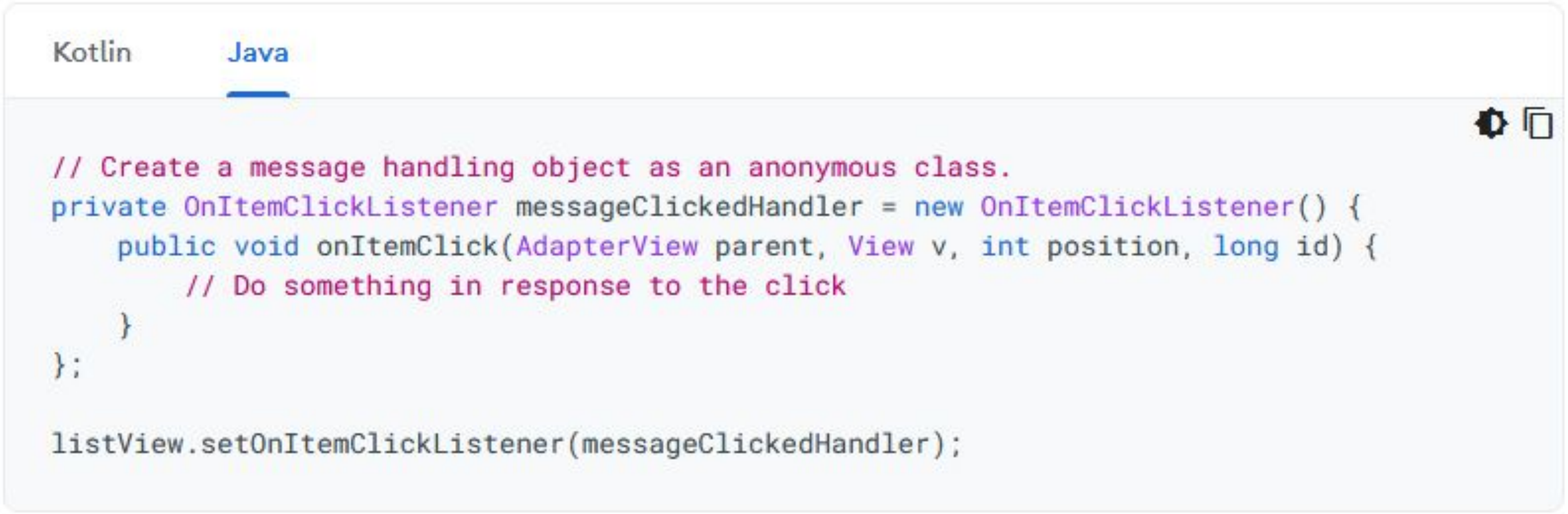
- When you instantiate the SimpleCursorAdapter, pass the layout to use for each result, the Cursor containing the results, and these two arrays:


Kotlin Java

```java
SimpleCursorAdapter adapter = new SimpleCursorAdapter(this,
        R.layout.person_name_and_number, cursor, fromColumns, toViews, 0);
ListView listView = getListView();
listView.setAdapter(adapter);
```

- The SimpleCursorAdapter then creates a view for each row in the Cursor using the provided layout by inserting each fromColumns item into the corresponding toViews view.

- **Handling click events**
    - You can respond to click events on each item in an AdapterView by implementing the AdapterView.OnItemClickListener interface. For example:



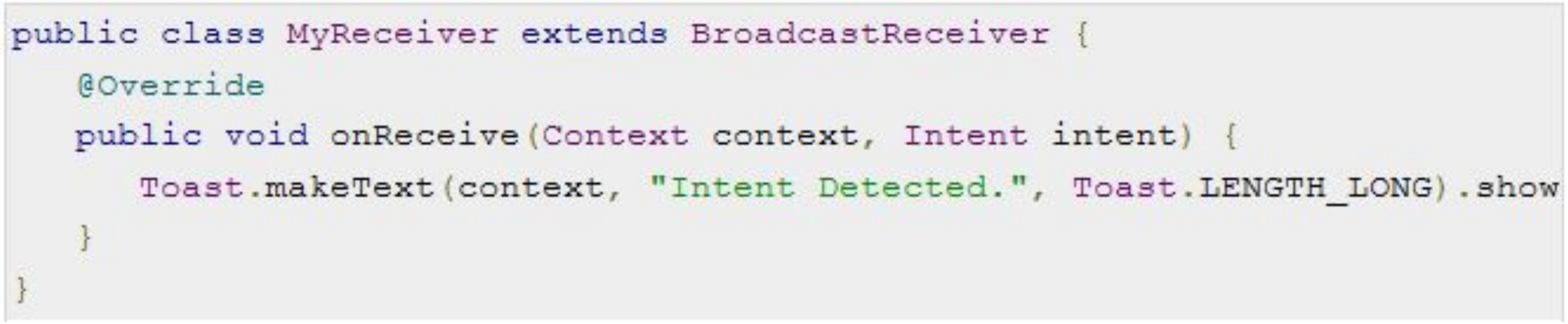
Kotlin Java

```
// Create a message handling object as an anonymous class.
private OnItemClickListener messageClickedHandler = new OnItemClickListener() {
    public void onItemClick(AdapterView parent, View v, int position, long id) {
        // Do something in response to the click
    }
};

listView.setOnItemClickListener(messageClickedHandler);
```

- Broadcast Receivers
  - 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).
  - 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, broadcasts can be used as a messaging system across apps and outside of the normal user flow. But it is slow.

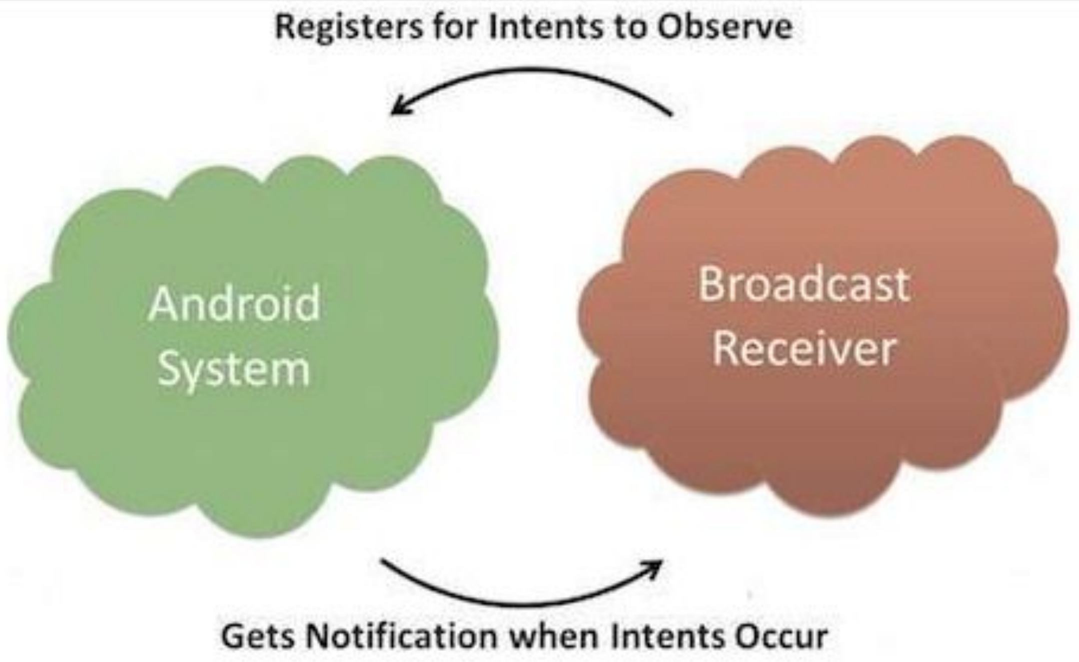
- There are following two important steps to make BroadcastReceiver works for the system broadcasted intents −
  - Creating the Broadcast Receiver.
  - Registering Broadcast Receiver

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



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

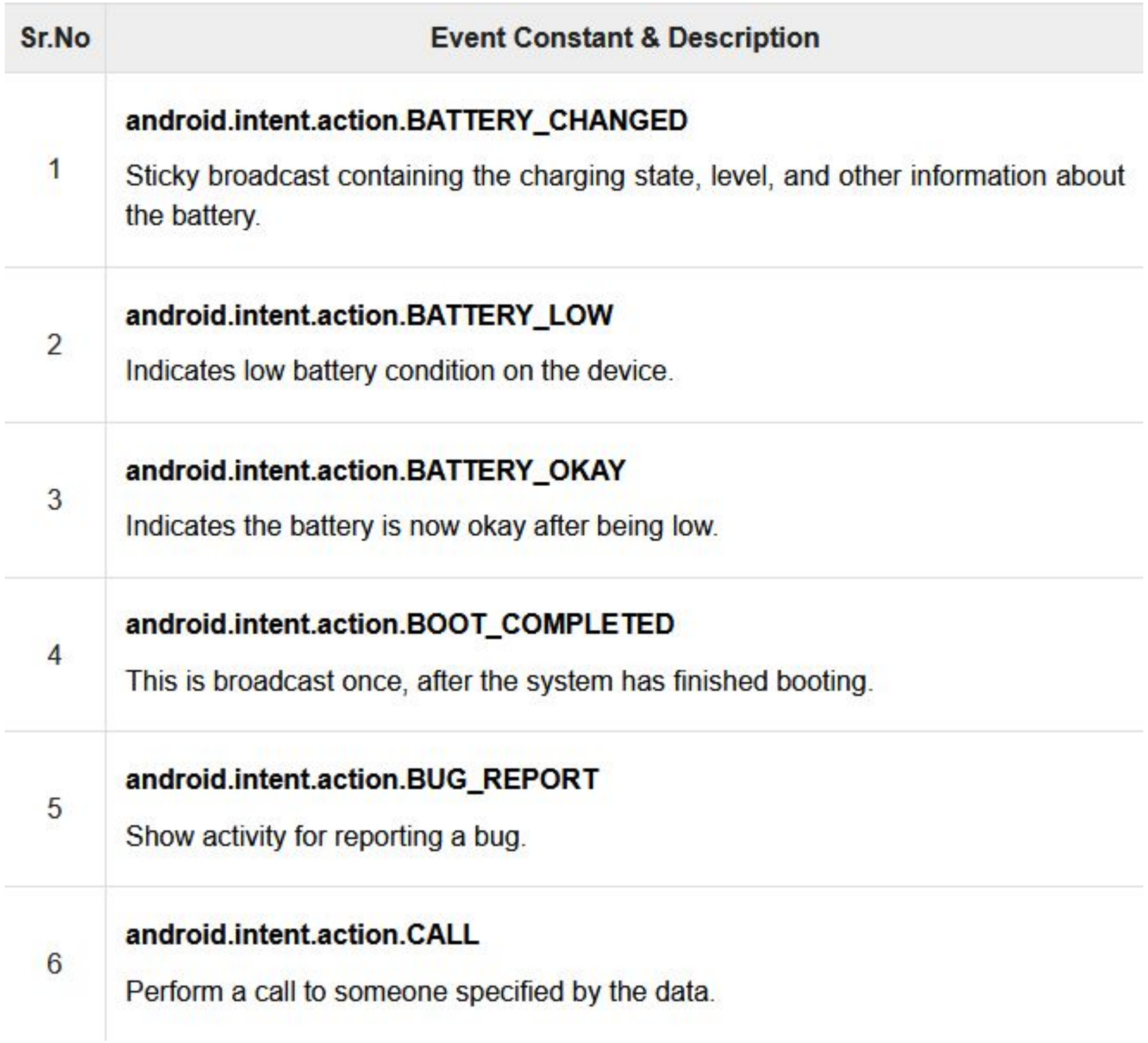
- **Registering Broadcast Receiver**
  - An application listens for specific broadcast intents by registering a broadcast receiver in *AndroidManifest.xml* file.
  - Consider we are going to register *MyReceiver* for system generated event ACTION_BOOT_COMPLETED which is fired by the system once the Android system has completed the boot process.
  - Now whenever your Android device gets booted, it will be intercepted by BroadcastReceiver *MyReceiver* and implemented logic inside *onReceive()* will be executed.


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

Broadcast-Receiver

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

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

   </receiver>
</application>
```

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

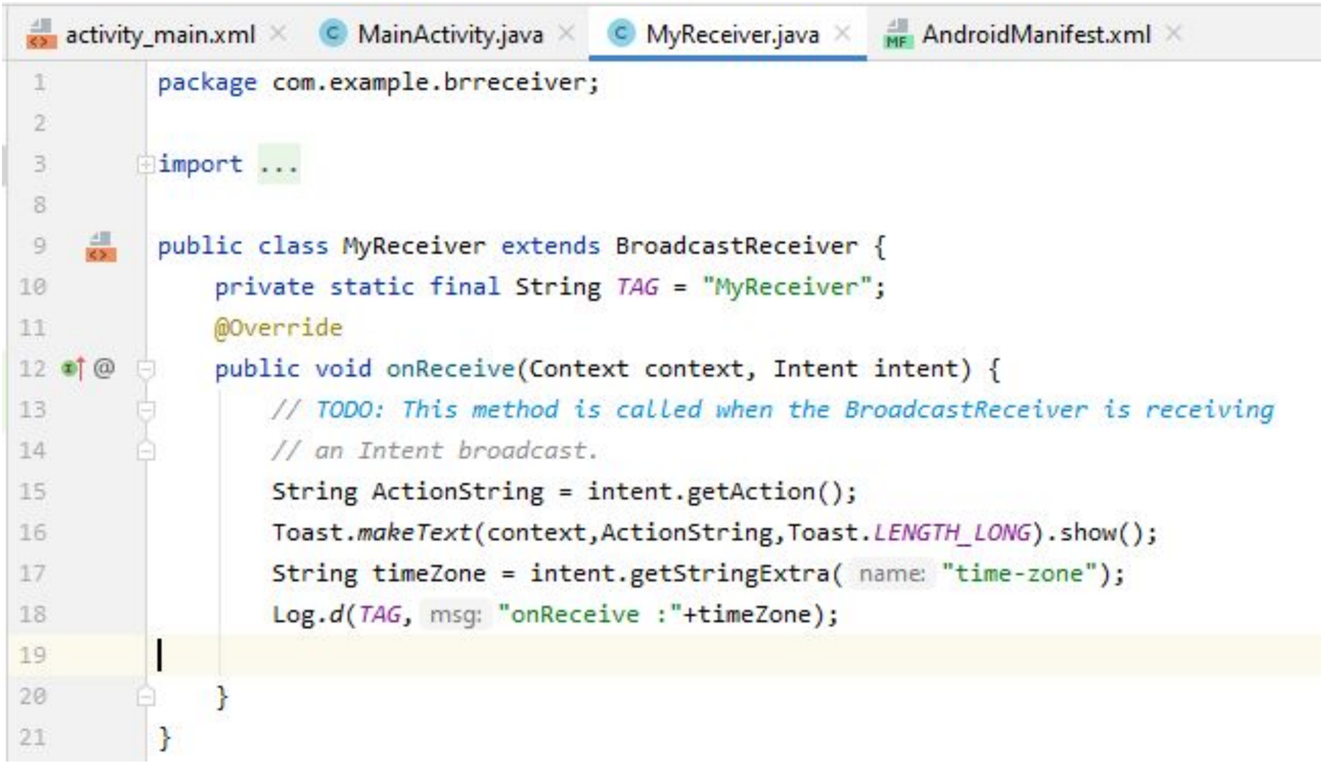


Sr.No	Event Constant & Description
1	**android.intent.action.BATTERY_CHANGED** Sticky broadcast containing the charging state, level, and other information about the battery.
2	**android.intent.action.BATTERY_LOW** Indicates low battery condition on the device.
3	**android.intent.action.BATTERY_OKAY** Indicates the battery is now okay after being low.
4	**android.intent.action.BOOT_COMPLETED** This is broadcast once, after the system has finished booting.
5	**android.intent.action.BUG_REPORT** Show activity for reporting a bug.
6	**android.intent.action.CALL** Perform a call to someone specified by the data.

7	**android.intent.action.CALL_BUTTON** The user pressed the "call" button to go to the dialer or other appropriate UI for placing a call.
8	**android.intent.action.DATE_CHANGED** The date has changed.
9	**android.intent.action.REBOOT** Have the device reboot.

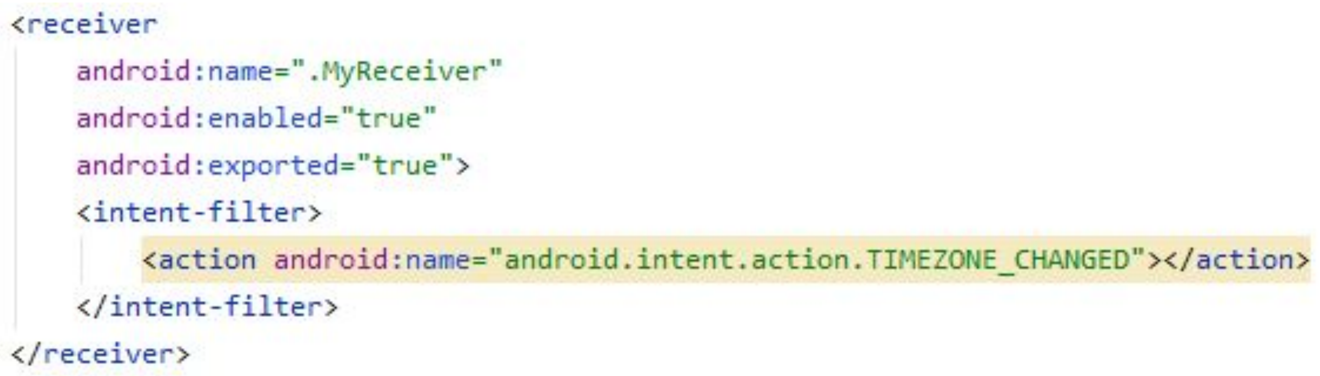
- There are two ways user can create broadcast receivers
  - Manifest declared context receiver
  - Context declared context receiver

- Manifest declared context receiver
- MyReceiver.java



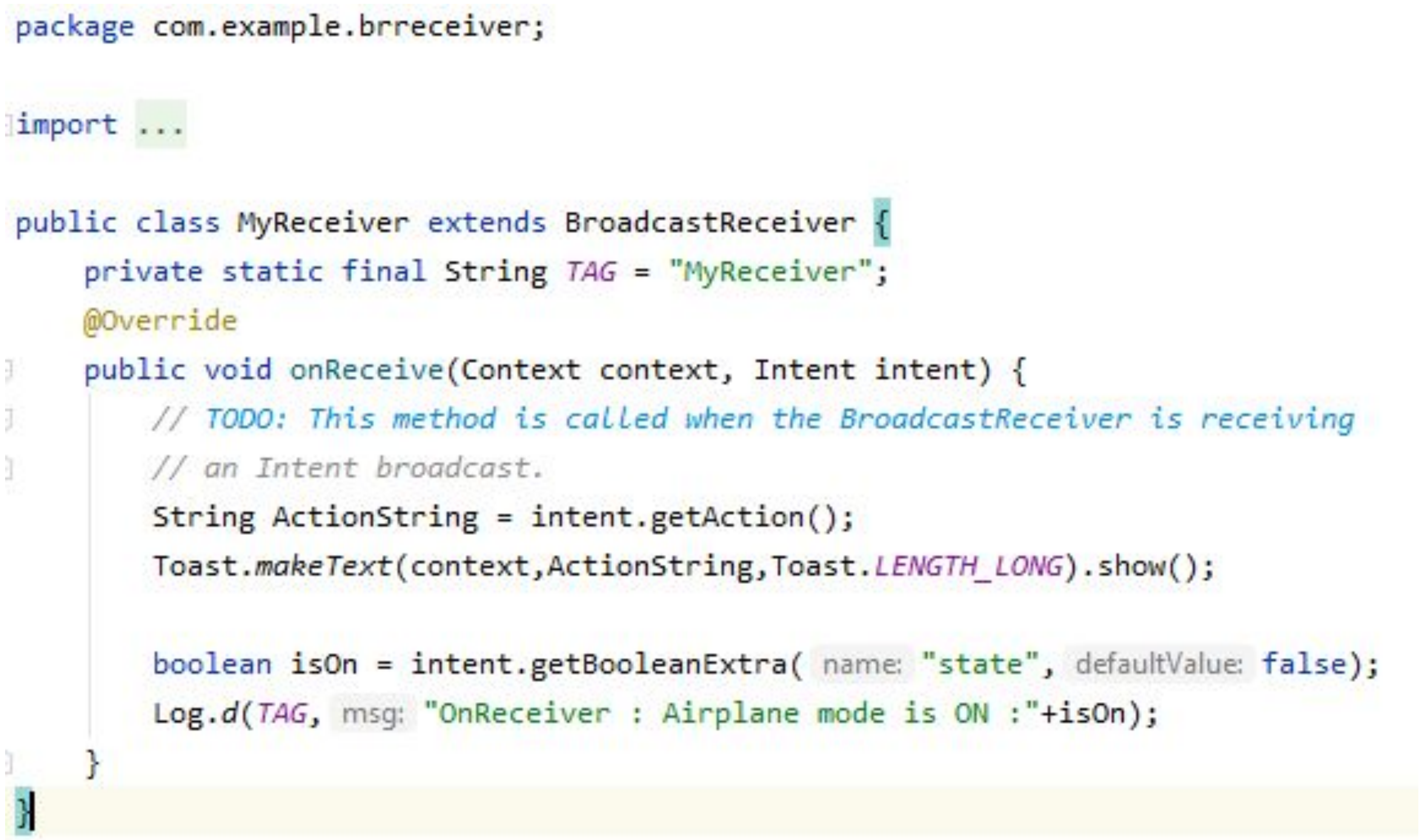
```java
package com.example.brreceiver;

import ...

public class MyReceiver extends BroadcastReceiver {
    private static final String TAG = "MyReceiver";
    @Override
    public void onReceive(Context context, Intent intent) {
        // TODO: This method is called when the BroadcastReceiver is receiving
        // an Intent broadcast.
        String ActionString = intent.getAction();
        Toast.makeText(context,ActionString,Toast.LENGTH_LONG).show();
        String timeZone = intent.getStringExtra( name: "time-zone");
        Log.d(TAG, msg: "onReceive :"+timeZone);

    }
}
```

- AndroidManifest.xml



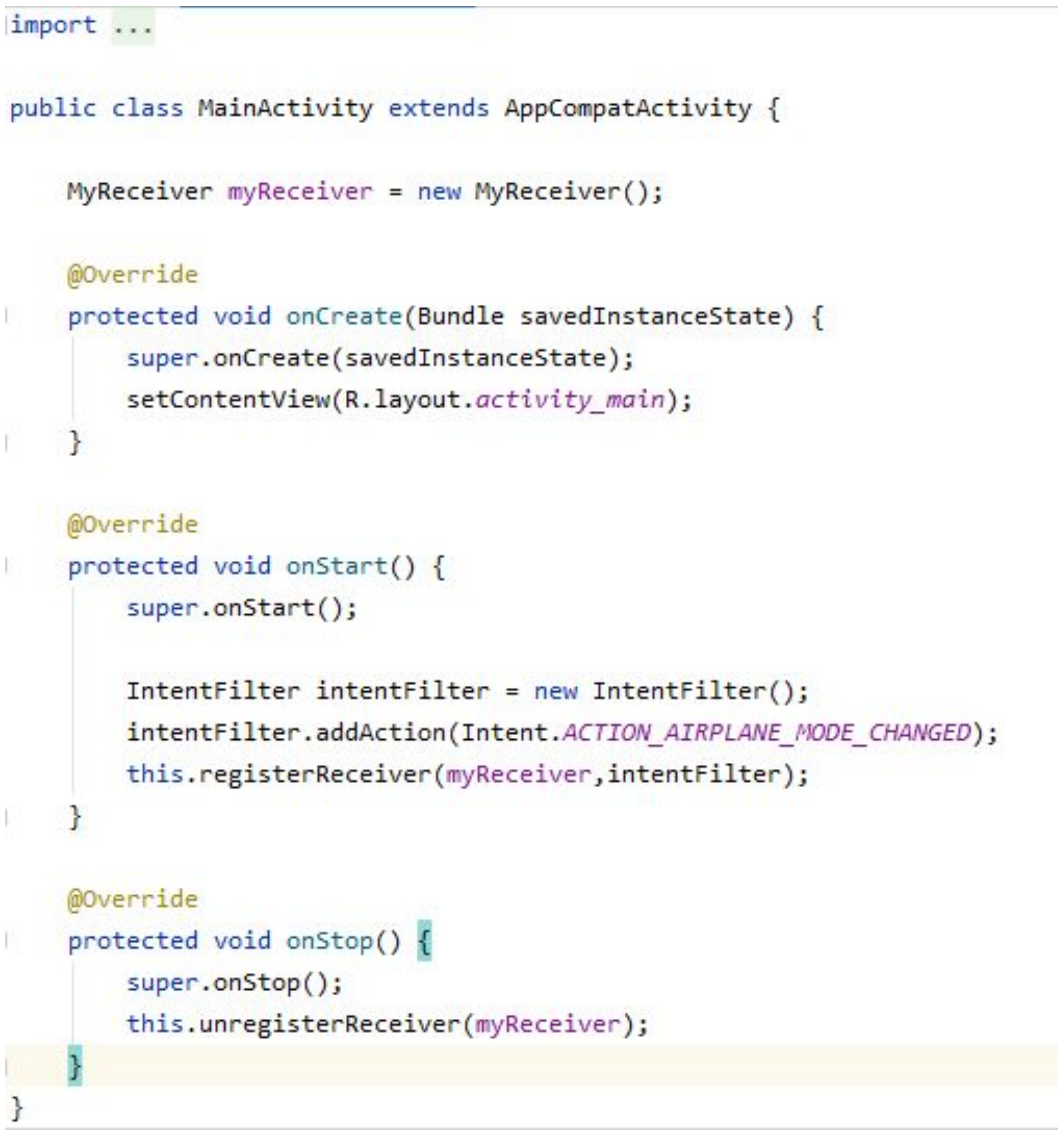
```xml
<receiver
    android:name=".MyReceiver"
    android:enabled="true"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.TIMEZONE_CHANGED"></action>
    </intent-filter>
</receiver>
```

- Content Declared broadcast Receiver
- MyReceiver.java



```
package com.example.brreceiver;

import ...

public class MyReceiver extends BroadcastReceiver {
    private static final String TAG = "MyReceiver";
    @Override
    public void onReceive(Context context, Intent intent) {
        // TODO: This method is called when the BroadcastReceiver is receiving
        // an Intent broadcast.
        String ActionString = intent.getAction();
        Toast.makeText(context,ActionString,Toast.LENGTH_LONG).show();

        boolean isOn = intent.getBooleanExtra( name: "state", defaultValue: false);
        Log.d(TAG, msg: "OnReceiver : Airplane mode is ON :"+isOn);
    }
}
```

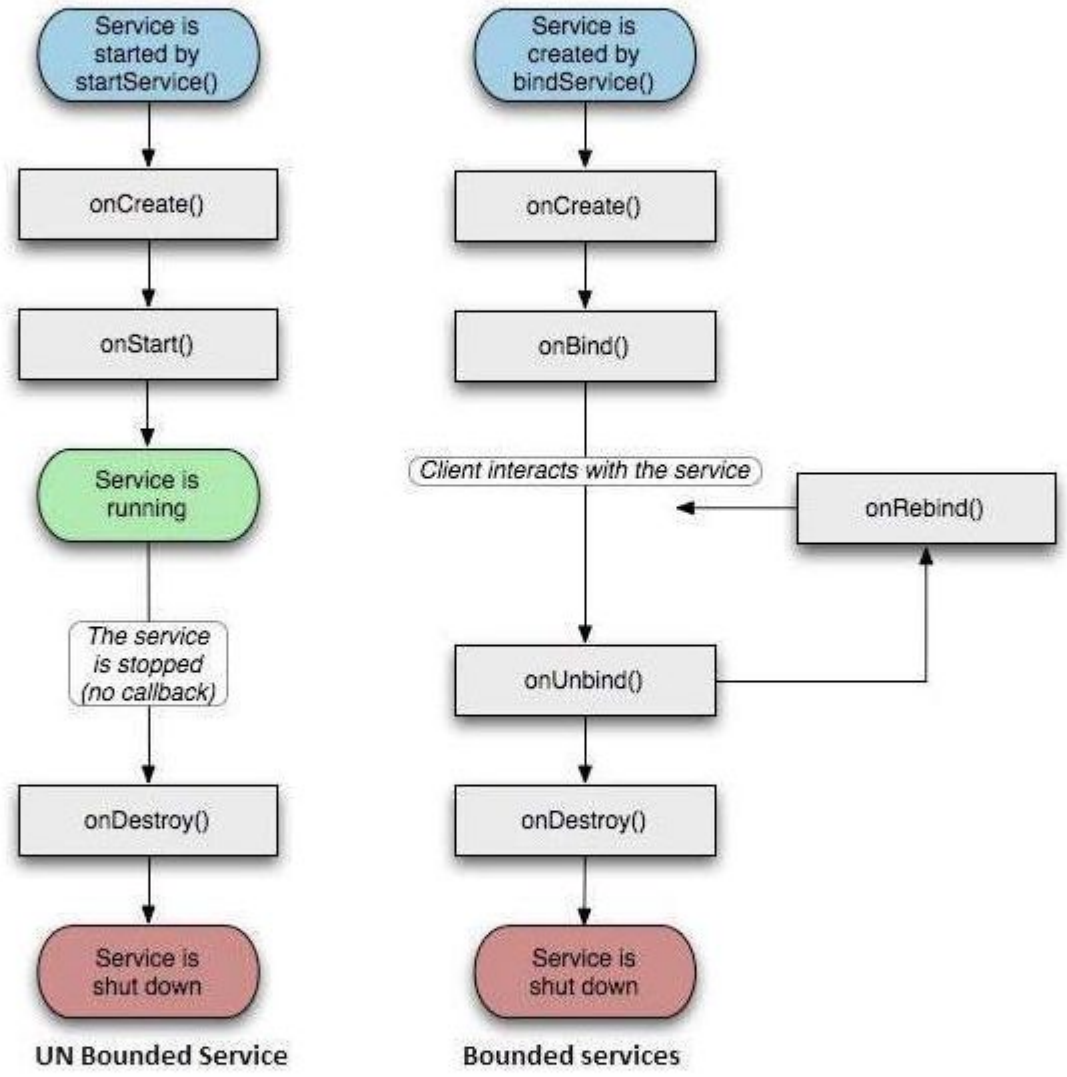
- MainActivity.java



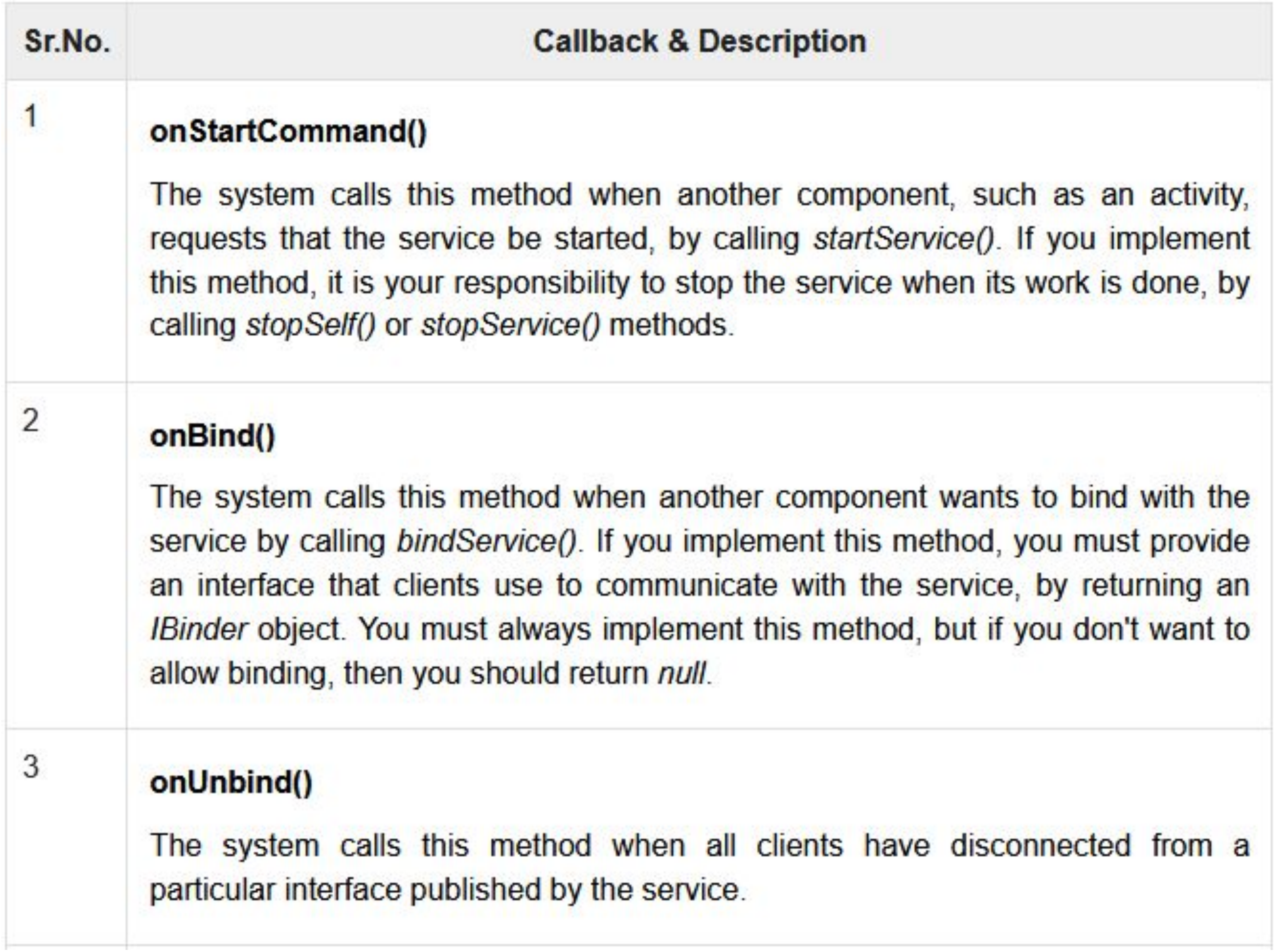
```
import ...

public class MainActivity extends AppCompatActivity {

    MyReceiver myReceiver = new MyReceiver();

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

    @Override
    protected void onStart() {
        super.onStart();

        IntentFilter intentFilter = new IntentFilter();
        intentFilter.addAction(Intent.ACTION_AIRPLANE_MODE_CHANGED);
        this.registerReceiver(myReceiver,intentFilter);
    }

    @Override
    protected void onStop() {
        super.onStop();
        this.unregisterReceiver(myReceiver);
    }
}
```

- 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.
    - **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():



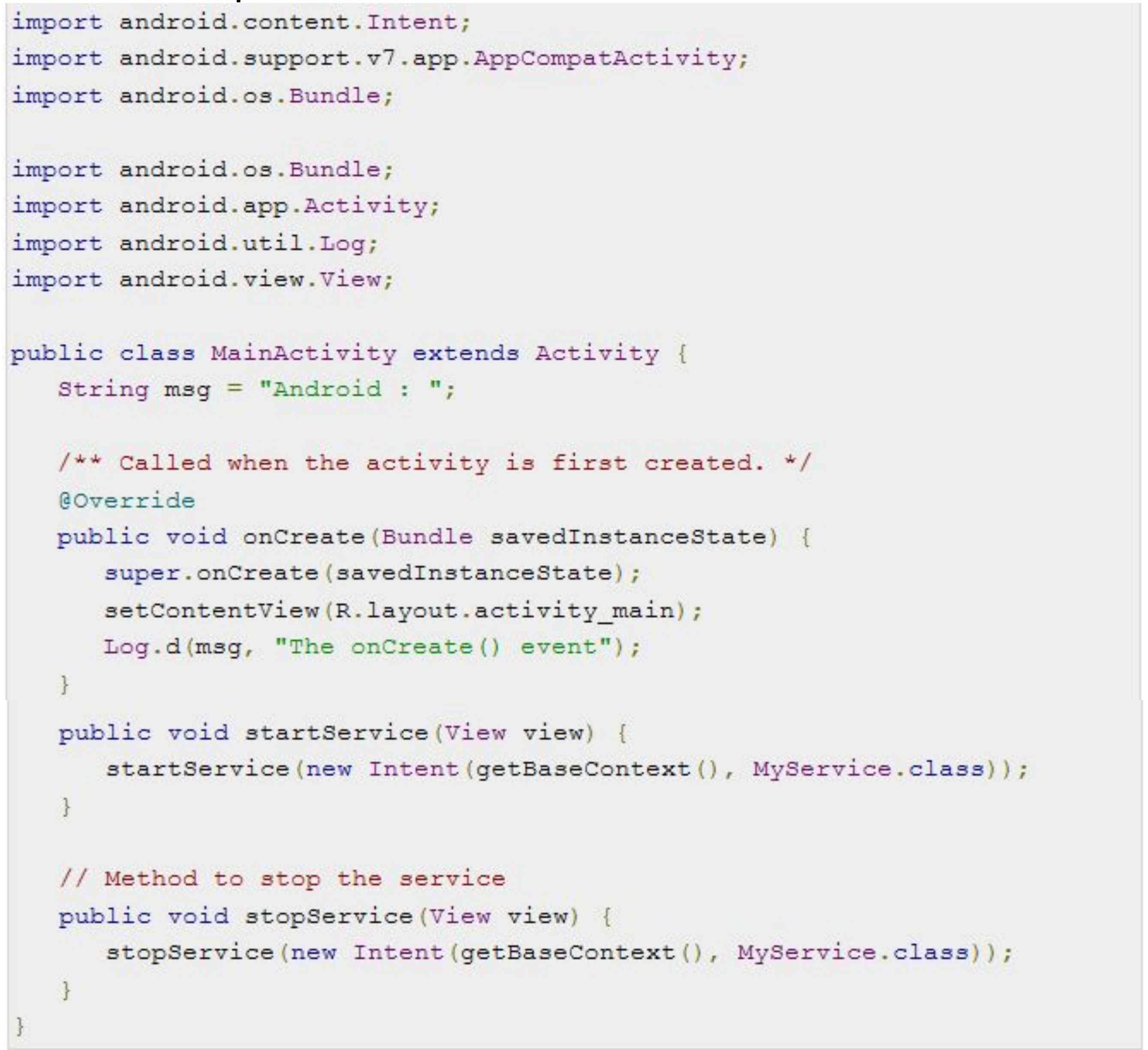
- To create an service, you create a Java class that extends the Service base class or one of its existing subclasses.
- The **Service** base class defines various callback methods and the most important are given below.



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.

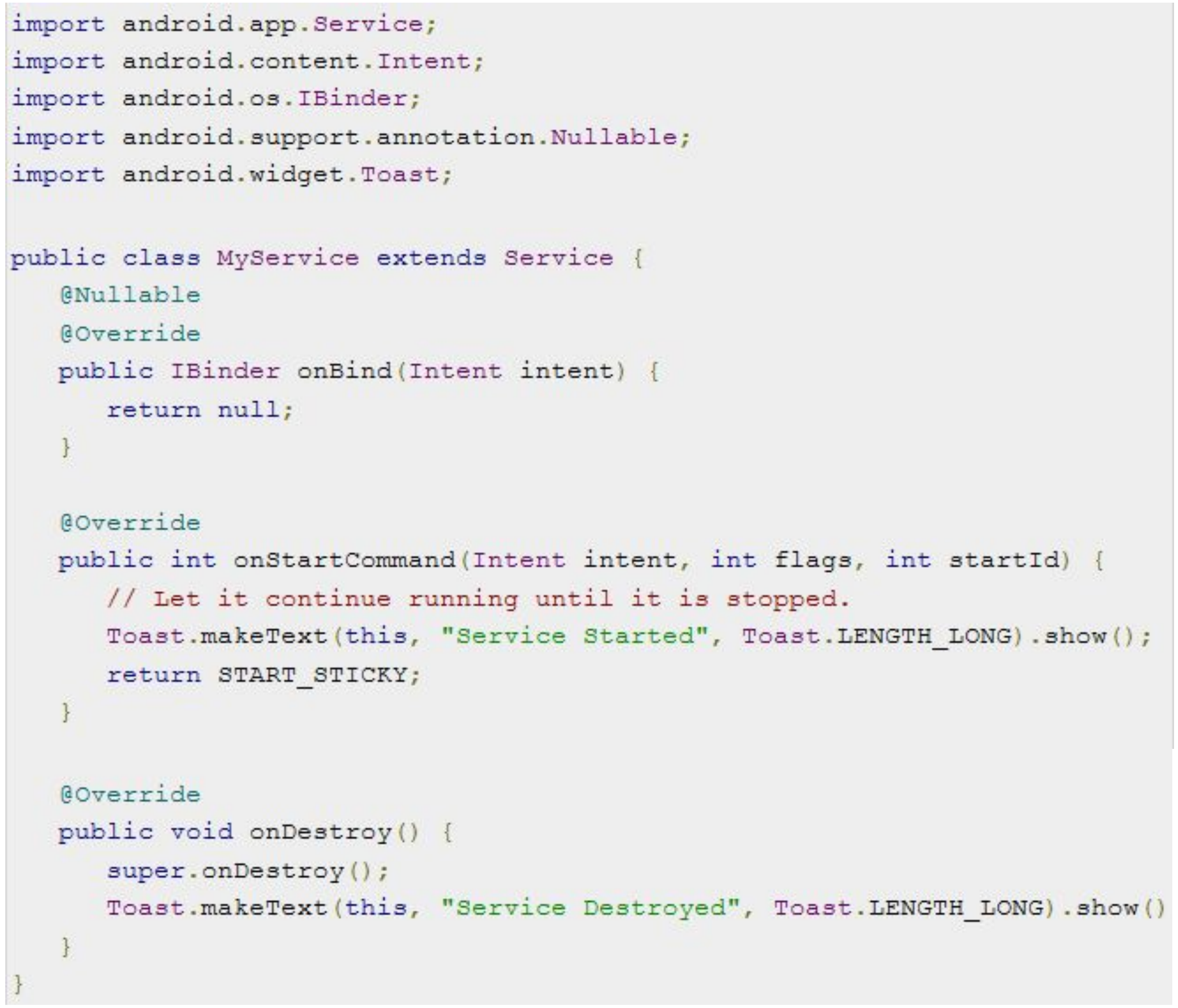
4	**onRebind()** The system calls this method when new clients have connected to the service, after it had previously been notified that all had disconnected in its *onUnbind(Intent)*.
5	**onCreate()** The system calls this method when the service is first created using *onStartCommand()* or *onBind()*. 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.

- 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.



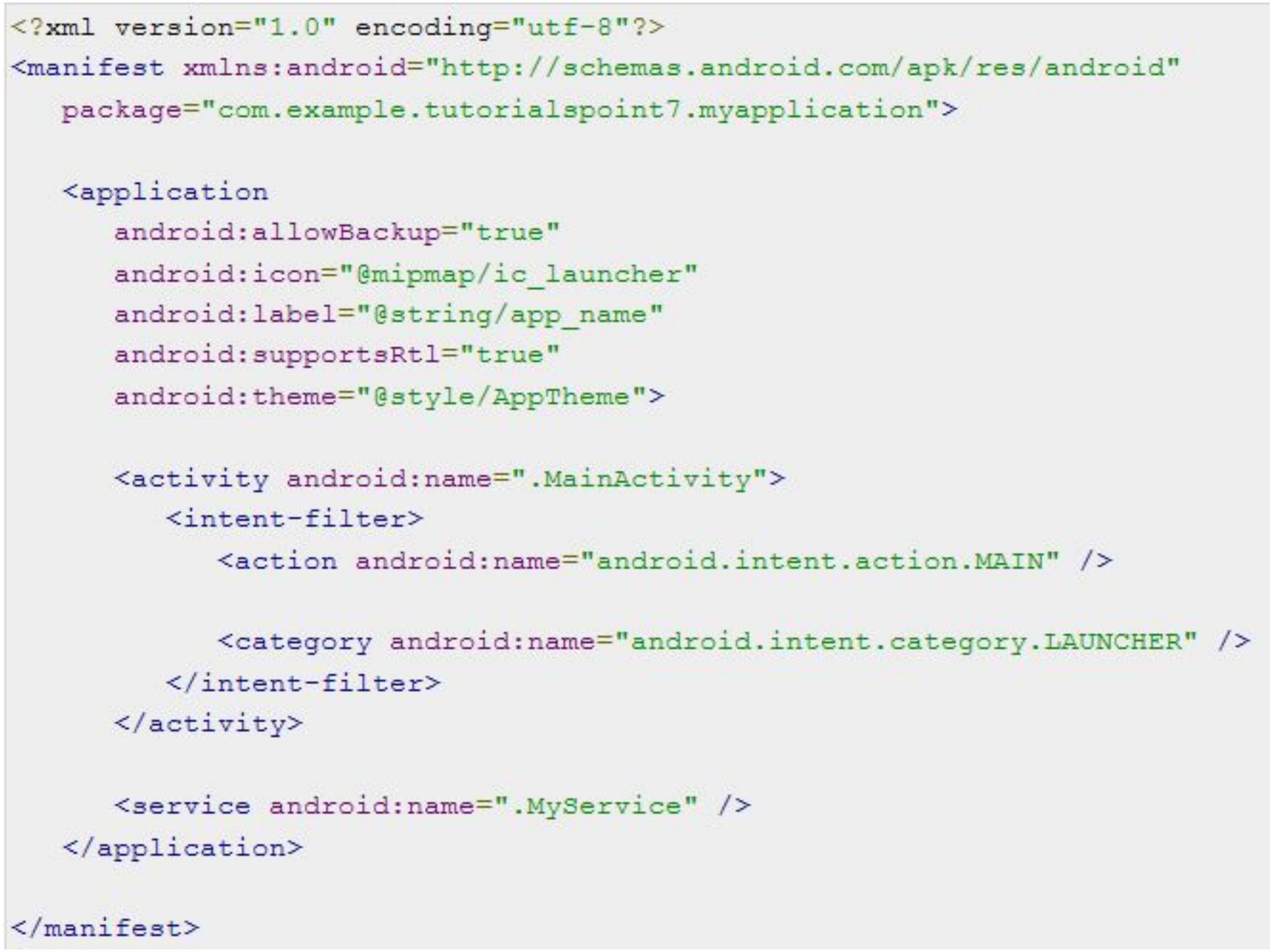
```java
import android.content.Intent;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

import android.os.Bundle;
import android.app.Activity;
import android.util.Log;
import android.view.View;

public class MainActivity extends Activity {
   String msg = "Android : ";

   /** Called when the activity is first created. */
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      Log.d(msg, "The onCreate() event");
   }

   public void startService(View view) {
      startService(new Intent(getBaseContext(), MyService.class));
   }

   // Method to stop the service
   public void stopService(View view) {
      stopService(new Intent(getBaseContext(), MyService.class));
   }
}
```

- Following is the content of **MyService.java**. This file can have implementation of one or more methods associated with Service based on requirements.



```java
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.support.annotation.Nullable;
import android.widget.Toast;

public class MyService extends Service {
   @Nullable
   @Override
   public IBinder onBind(Intent intent) {
      return null;
   }

   @Override
   public int onStartCommand(Intent intent, int flags, int startId) {
      // Let it continue running until it is stopped.
      Toast.makeText(this, "Service Started", Toast.LENGTH_LONG).show();
      return START_STICKY;
   }

   @Override
   public void onDestroy() {
      super.onDestroy();
      Toast.makeText(this, "Service Destroyed", Toast.LENGTH_LONG).show()
   }
}
```

- Following will the modified content of *AndroidManifest.xml* file.



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

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

      <activity android:name=".MainActivity">
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />

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

      <service android:name=".MyService" />
   </application>

</manifest>
```

- 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.
  - Most importantly you can configure a content provider to allow other applications to securely access and modify your app data

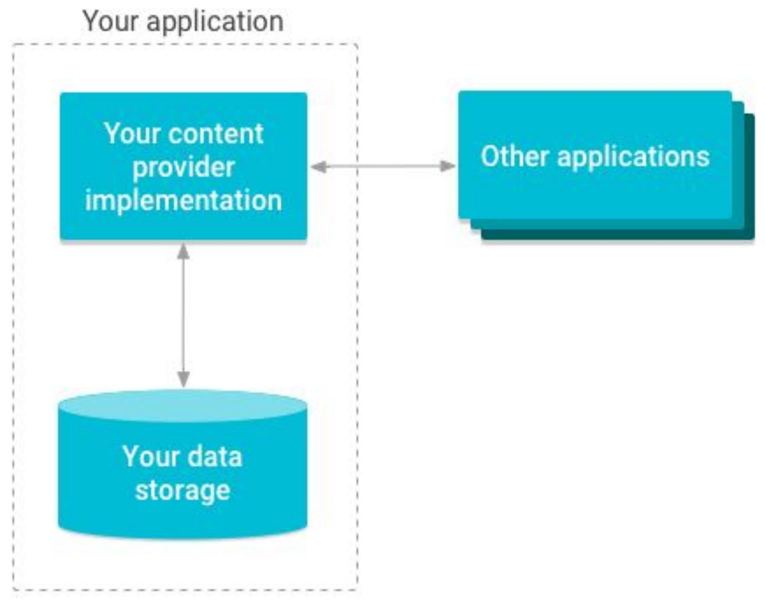


Figure 1. Overview diagram of how content providers manage access to storage.

- Use content providers if you plan to share data.
- If you don't plan to share data, you may still use them because they provide a nice abstraction, but you don't have to.
- This abstraction allows you to make modifications to your application data storage implementation without affecting other existing applications that rely on access to your data.
- In this scenario only your content provider is affected and not the applications that access it.

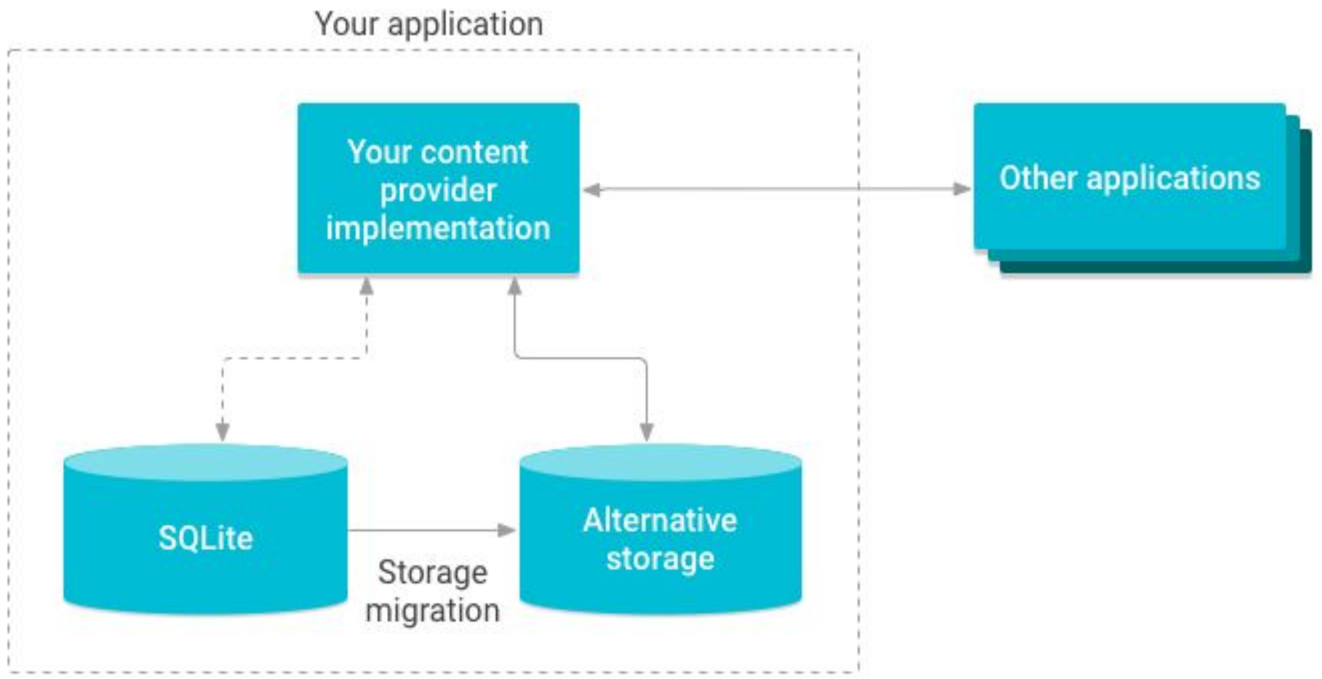


Figure 2. Illustration of migrating content provider storage.

- A number of other classes rely on the ContentProvider class:

    - AbstractThreadedSyncAdapter
    - CursorAdapter
    - CursorLoader

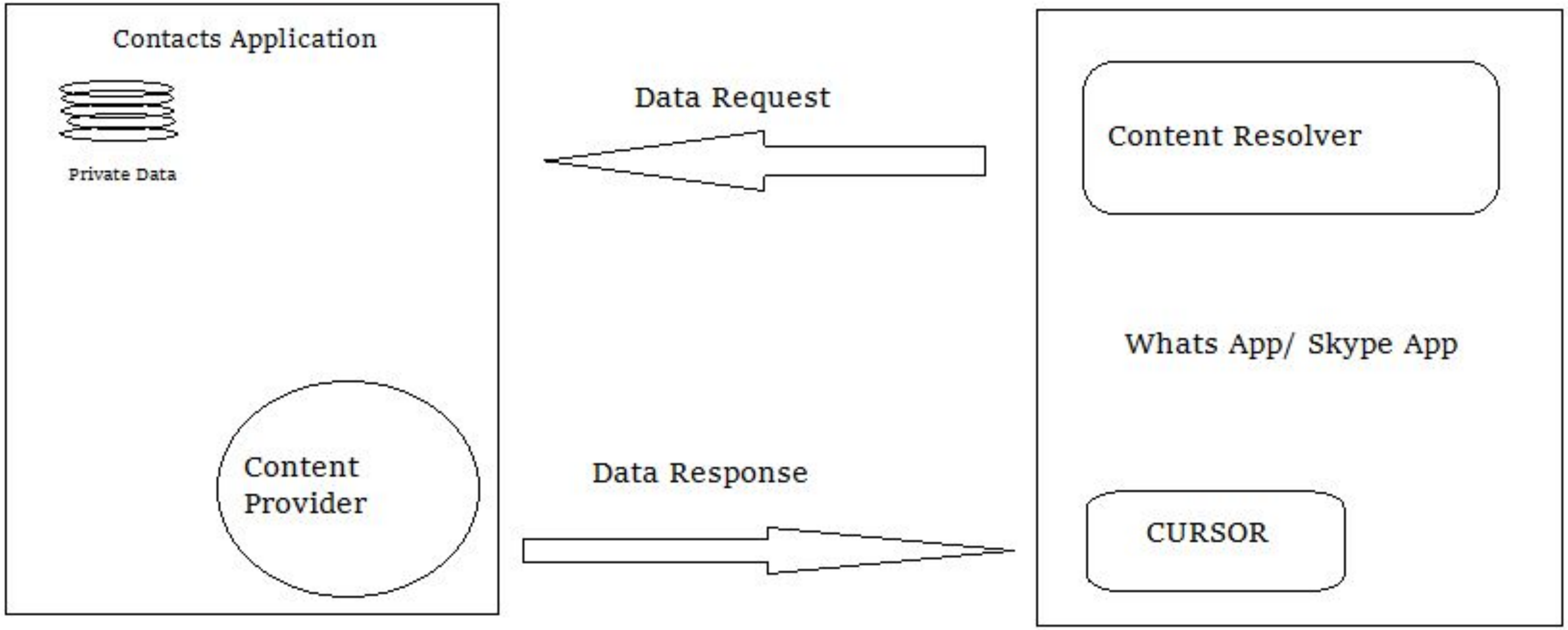
    - If you are making use of any of these classes you also need to implement a content provider in your application. Note that when working with the sync adapter framework you can also create a stub content provider as an alternative.

- In addition, you need your own content provider in the following cases:
    - You want to implement custom search suggestions in your application
    - You need to use a content provider to expose your application data to widgets
    - You want to copy and paste complex data or files from your application to other applications

- The Android framework includes content providers that manage data such as audio, video, images, and personal contact information.

- A content provider can be used to manage access to a variety of data storage sources, including both structured data, such as a SQLite relational database, or unstructured data such as image files.

- Content Provider / Content Resolver:



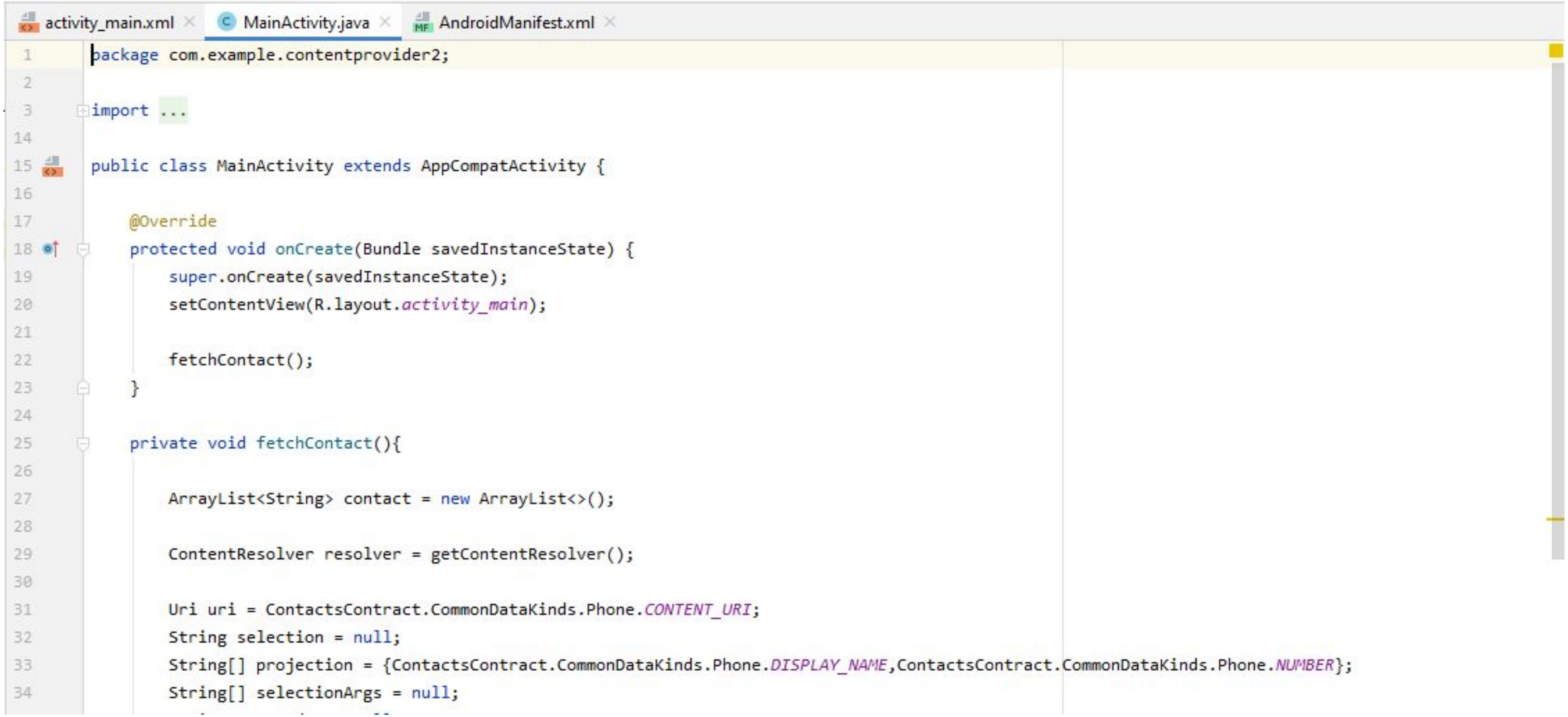
- Content Resolver API used by application2 (WhatsApp/Skype) to hit the application1(Contacts) data
- Content Provider in application1(Contacts) will respond with a format called CURSOR.
- Application1 exposes its data in a secure manner using Content Provider.
- Application2 doesn’t bother how data is stored.
- The data may be stored in
  - Database
  - JSON file
  - XML file
  - file

- The context object contains method getContentResolver() – which returns the instance of a contentResolver

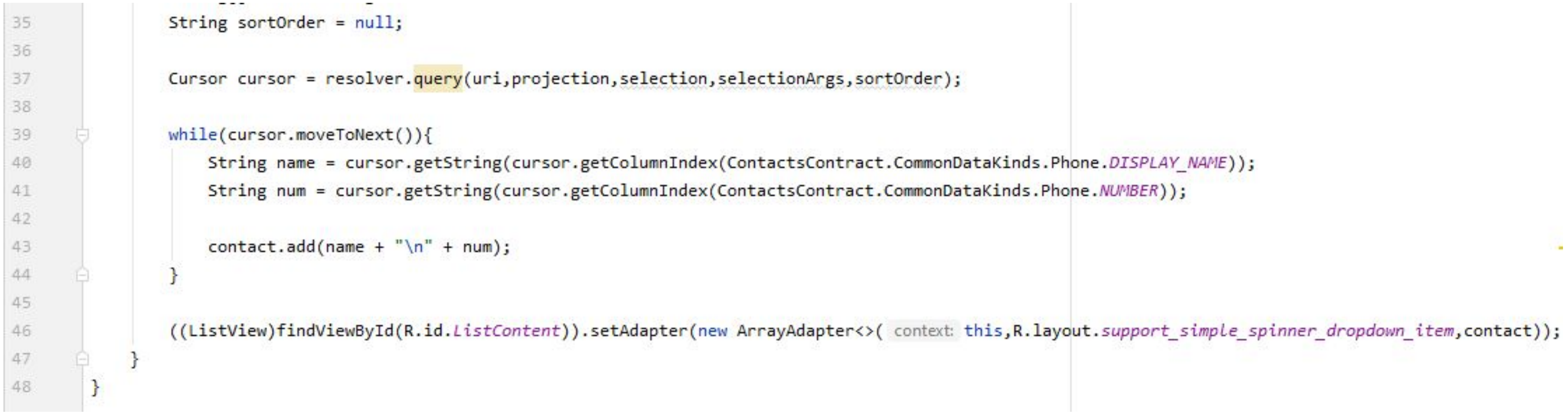
- To hit the correct Content Resolver you should know the correct – (URI) Uniform Resource Identifier

- Content Resolver provider contain methods using which user can perform
  - CRUD operations.
    - Create
    - Retrieve
    - Update
    - Delete
  - Batch operations.

- If content provider provides the data in tabular format CURSOR is an API which is used traverse through the table.

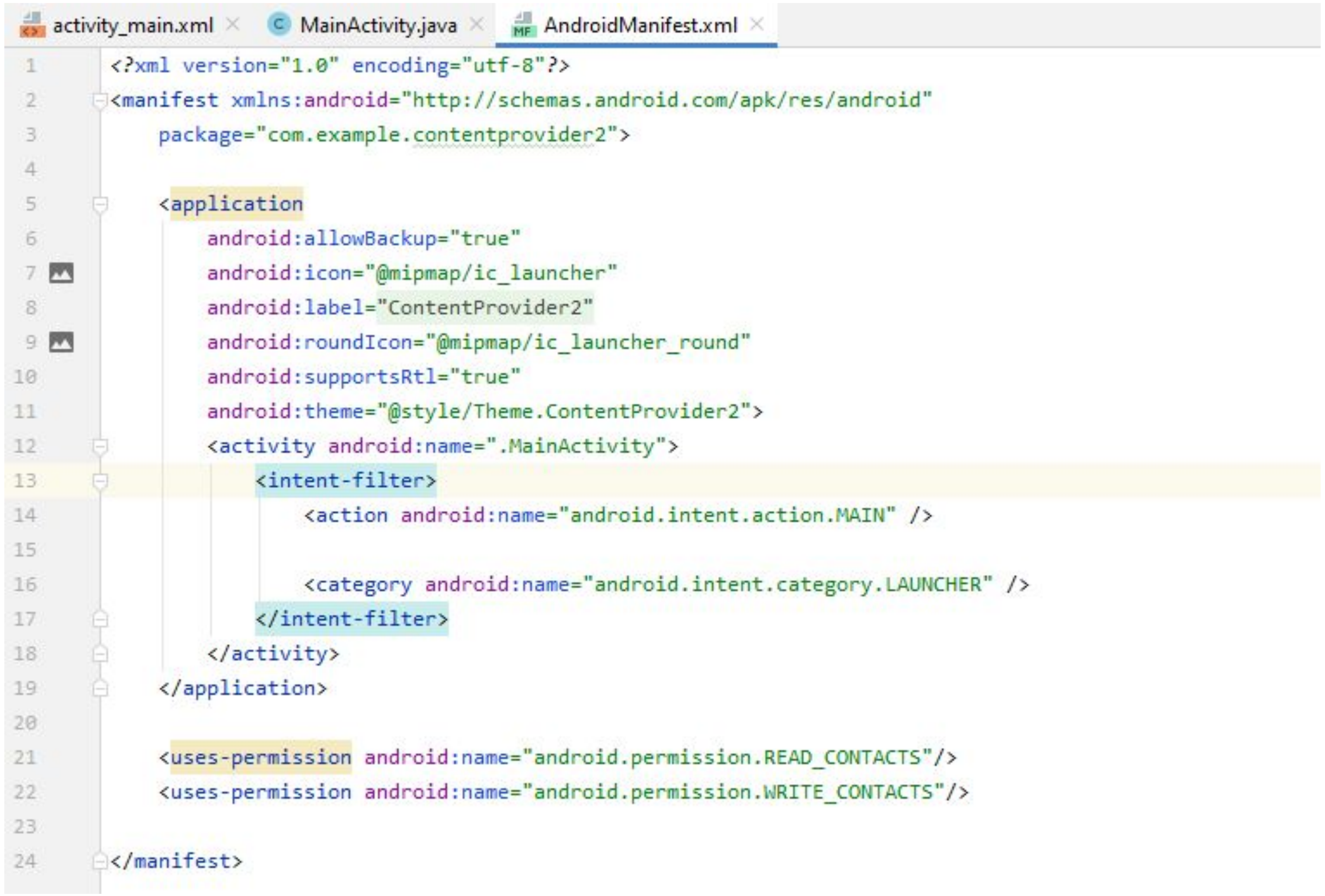


```
package com.example.contentprovider2;

import ...

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        fetchContact();
    }

    private void fetchContact(){

        ArrayList<String> contact = new ArrayList<>();

        ContentResolver resolver = getContentResolver();

        Uri uri = ContactsContract.CommonDataKinds.Phone.CONTENT_URI;
        String selection = null;
        String[] projection = {ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME,ContactsContract.CommonDataKinds.Phone.NUMBER};
        String[] selectionArgs = null;
```



```
        String sortOrder = null;

        Cursor cursor = resolver.query(uri,projection,selection,selectionArgs,sortOrder);

        while(cursor.moveToNext()){
            String name = cursor.getString(cursor.getColumnIndex(ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME));
            String num = cursor.getString(cursor.getColumnIndex(ContactsContract.CommonDataKinds.Phone.NUMBER));

            contact.add(name + "\n" + num);
        }

        ((ListView)findViewById(R.id.ListContent)).setAdapter(new ArrayAdapter<>( context: this,R.layout.support_simple_spinner_dropdown_item,contact));
    }
}
```

- Androidmanifest.xml
  - Add the following code in uses-permission:
  - android.permission.READ_CONTACTS
  - android.permission.WRITE_CONTACTS



```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.contentprovider2">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="ContentProvider2"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.ContentProvider2">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

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

    <uses-permission android:name="android.permission.READ_CONTACTS"/>
    <uses-permission android:name="android.permission.WRITE_CONTACTS"/>

</manifest>
```

- Thank you