Mobile Development

Riccardo Caprile

March 2023

1 Introduction to Android

1.1 What is an Android app?

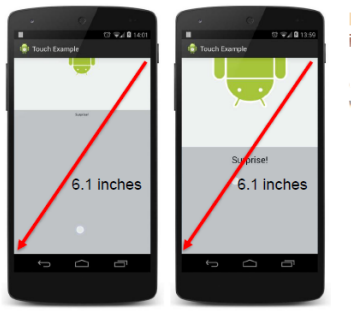
App that uses one or more interactive screens , written using Java and Kotlin (logic) and XML (UI). Uses the Android Software Development Kit and Android libraries and Android Application Framework.

1.2 Challenges of Android Development

1.2.1 Multiple screen sizes and resolutions

Challenge , as a developer , is to design UI elements that work on all devices.

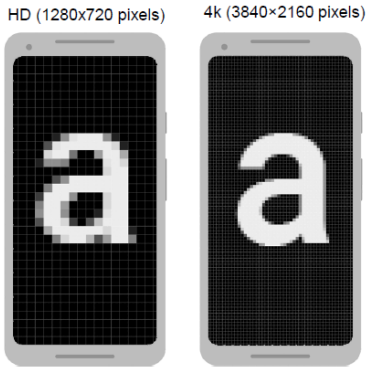
Screen Property : Density



If you use fixed measures in your app , you can obtain the result on the right on the device you are working. But you can have the situation on the left on a device with an higher pixel density.

Example of two screens of the same size may have a different number of pixels.

To preserve the visible size of your UI on screens with different densities , you must design your UI using **density-independent pixels (dp)** as your unit of measurement.



dp is a virtual pixel unit that is roughly equal to one pixel on a medium-density screen.

1.2.2 Performance

Make your apps responsive and smooth. Main aspects :

- How fast it runs
- How easily it connects to the network
- How well it manages battery and memory usage

Performances are affected by various factors :

- Battery level (low battery , low performance)
- Multimedia content (HD content reduce overall performances)
- Internet access

1.2.3 Security

Keep source code and user data safe.

Protect critical user information such as login and passwords, secure your communication channel to protect data in transit across the internet , optimize the code , remove unused resources , classes , fields and methods.

1.2.4 Compatibility

Run well on older platform versions. Not focus only on the most recent Android version. Not all users may have upgraded their devices.

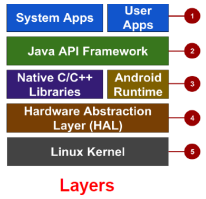
1.3 App Building Blocks

- **Resources** : layouts , images , strings and colors (XML)
- **Components** : activities , services and helper classes as Java and Kotlin
- **Manifest** : information about the app for the runtime
- **Build configuration** : Gradle config files

2 Android Platform Architecture

2.1 Android Stack

1. System and user apps
2. Android OS API Java framework
3. Expose native APIs; run apps
4. Expose device hardware capabilities
5. Linux Kernel



2.1.1 System and user apps

System apps have **no special status**

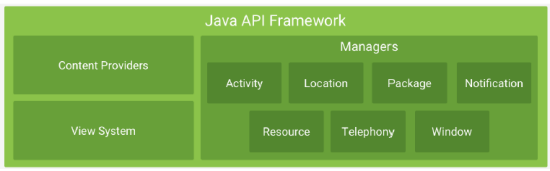
System app **provide key capabilities** to app developers (email , SMS , calendars , internet browsing).

For example , your app can use a system app to deliver a SMS message.

2.1.2 JAVA Api Framework

The entire feature-set of the Android OS is available to you through APIs written in the Jave language.

- View System : to create UI screens
- Notification Manager : to display custom alerts in the status bar
- Activity Manager : for app life cycles and navigation



2.2 Android runtime

When we build our app and generate APK , part of that APK are .dex files.

When a user runs our app the bytecode written in .dex files is translated by Android Runtime into the machine code.

Each app runs in its own process with its own instance of the Android Runtime(ART).

For devices running Android version 5.0 or higher.

Android also includes a set of core runtime libraries that provide most of the functionality of the Java programming language.

2.3 C/C++ libraries

Core C/C++ libraries give access to core native Android system components and services.

2.4 Hardware Abstraction Layer

Standard interfaces that expose device hardware capabilities as libraries (Camera , Bluetooth module , sensors).

When a framework API makes a call to access device hardware , the Android system loads the library module for that hardware component.

2.5 Linux Kernel

The foundation of the Android platform is the Linux Kernel. Features include :

- Threading and low-level memory management : ART relies on the Linux kernel for underlying functionalities such as threading and low level memory management
- Security : Using a Linux kernel allows Android to take advantage of key security features
- Drivers : Using a Linux kernel allows device manufacturers to develop hardware drivers for a well-known kernel

3 Logging in Android

3.1 LogCat pane

Run pane reports some messages but cannot be configured...

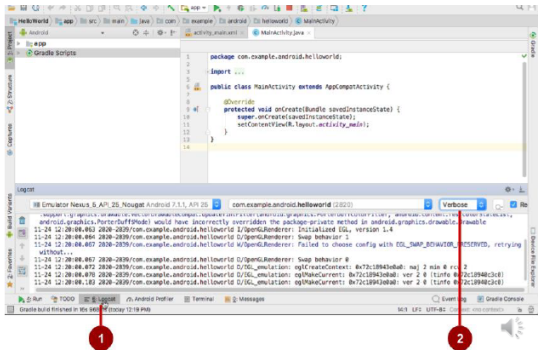
Instead, with the configurable **Logcat pane** it is possible to create custom view of :

System messages : such as when a garbage collection occurs

messages: added to the app with the **Log class**

It displays messages in real time and keeps a history so you can view older messages.

1. **Logcat** tab to show Logcat pane
2. Log level menu



3.1.1 Write log messages

The Log class allows to create log messages that appear in logcat.

Use the following log methods , listed in order from the highest to lowest priority:

- Log.e(String,String) (error)
- Log.w(String,String) (warning)
- Log.i(String,String) (information)
- Log.d(String,String) (debug)
- Log.v(String,String) (verbose)

private const val TAG = "MyActivity";
Log.i(Tag,"Message");

TAG: the first parameter should be a unique tag , a short string indicating the system component from which the message originates

3.1.2 Set the log level

It is possible to control how many messages appear in the logcat by setting the log level. In the Log level menu , select one of the following values:

- **Verbose** : Show all log messages
- **Debug**: Show debug log messages that are useful during development only
- **Info** : Show expected log messages for regular usage
- **Warn**: Show possible issues that are not yet errors
- **Error**: Show issues that have caused errors
- **Assert**: Show issues that the developer expects should never happen.

4 Views,View Groups and View Hierarchy

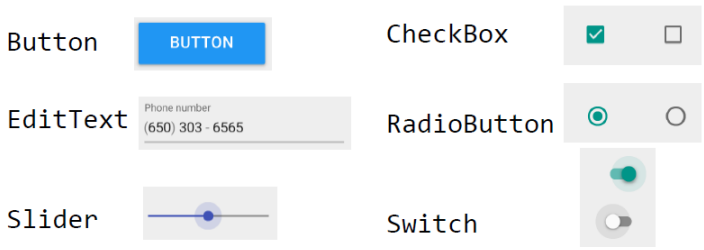
4.1 View

If you look at your mobile device, evey user interface element that you see is a **View**.

UI consists of a hierarchy of objects called views

View subclasses are basic user interface building blocks (Display text , Edit text, Buttons , menus , Scrollable, Group views).

4.2 Example of view subclasses



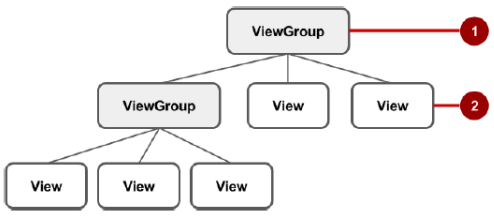
4.3 View Group and View Hierarchy

4.3.1 ViewGroup contains child views

View elements for a screen are organized in a hierarchy.

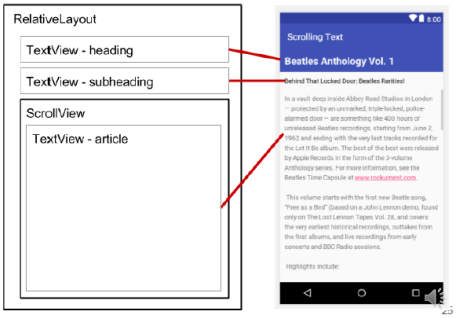
At the root of this hierarchy there is a ViewGroup(1)

ViewGroup can contain child View elements(2) or other ViewGroup



The following are commoonly used ViewGroups:

- ConstraintLayout : Positions UI elements using constraint connections to other elements and to the layout edges
- ScrollView: Contains one element and enebles scrolling



4.4 ViewGroups for layouts

Some ViewGroup groups are designated as layouts.

Organize child View elements in a specific way, they are typically used as the root ViewGroup

4.4.1 Common Layout Classes

A group of child View elements using constraints , edges and guidelines to control how the elements are positioned relative to other elements in the layout.

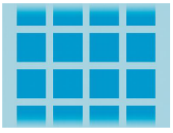
ConstraintLayout was designed also to make it easy to click and drag View elements in the layout editor.

ConstraintLayout

Linear Layout : A group of child View elements positioned and aligned horizontally or vertically

Grid Layout : A group that places its child View elements in a rectangular grid that can be scrolled

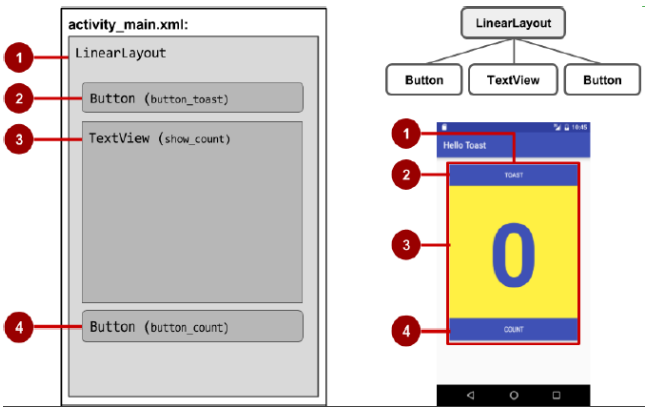
LinearLayout



GridLayout

View Class hierarchy is standard object-oriented class inheritance
Layout Hierarchy is how views are visually arranged.

4.4.2 View Hierarchy and Screen Layout



5 Layouts and Event Handling

5.1 Layout editor main toolbar

The figure above shows the top toolbar of the layout editor:

1. Select Design Surface : Select Design to display a color preview of the UI elements in your layout, or Blueprint to show only outlines of the elements. To see both panes side by side, select Design + Blueprint

2. Orientation in Editor : Select Portrait or Landscape to show the preview in a vertical or horizontal orientation. The orientation setting lets you preview the layout orientations without running the app on an emulator or device.

3. Device in Editor : Select the device type (phone/tablet, Android TV, or Android

4. API Version in Editor : Select the version of Android to use to show the preview

5. Theme in Editor : Select a theme (such as AppTheme) to apply to the preview

6. Locale in Editor : Select the language and locale for the preview. This list displays only the languages available in the string resources (see the documentation on localization for details on how to add languages).

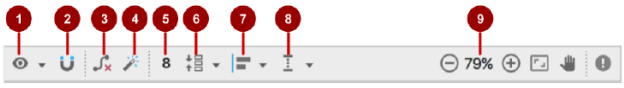
5.2 Preview layouts

It is possible to preview an app's layout with a horizontal orientatin , without having to run the app on an emulator or device.

Click Orientation in Editor button , choose Switch to Landscape or Switch to Portrait.

Preview layout with different devices : click device in Editor button , choose device.

5.3 ConstraintLayout toolbar in layout editor



1. Show : Select Show Constraints and Show Margins to show them in the preview, or to stop showing them.

2. Autoconnect : Enable or disable Autoconnect . With Autoconnect enabled, you can drag any element (such as a Button) to any part of a layout to generate constraints against the parent layout.

3. Pack : Clear All Constraints : Clear all constraints in the entire layout.

4. Infer Constraints : Create constraints by inference.

5. Default Margins : Set the default margins.

6. Pack : Pack or expand the selected elements.

7. Align : Align the selected elements.

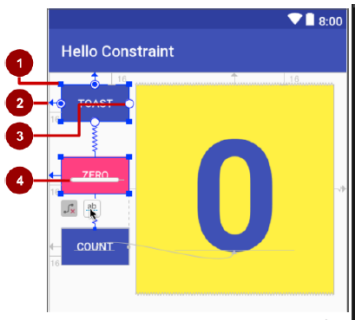
8. Guidelines : Add vertical or horizontal guidelines.

9. Zoom controls: Zoom in or out.

5.4 ConstraintLayout handles

A constraint is a connection or alignment to : another UI element, the parent layout , an invisible guideline.

Each constraint appears as a line extending from a circular handle.



1. Resizing square handle
2. Constraint line and handle . In the figure, the constraint aligns the left side of the Toast Button to the left side of the layout.
3. Constraint handle without a constraint line.
4. Baseline handle . The baseline handle aligns the text baseline of an element to the text baseline of another element.

To create a constraint : click a constraint handle (shown as a circle on each side of an element) , drag the circle to another constraint handle or to a parent boundary. A zigzag line represents the constraint.

5.5 Event Handling

5.5.1 Events

Something that happens.

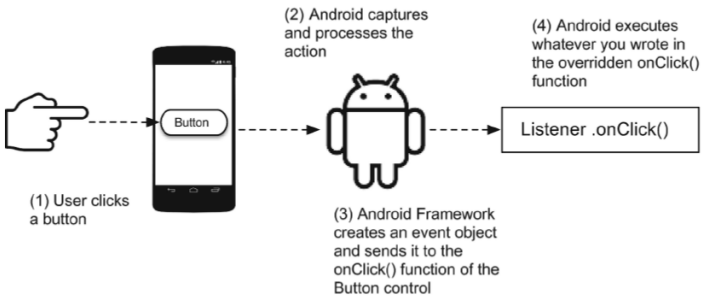
In UI: click,tap,drag

Device: DetectedActivity such as walking,driving,tilting.

5.5.2 Event Handlers

Methods that do something in response to an event (click or tap).

- An **event handler** is a method triggered by a specific event and that does something in response to such event.
- An **event listener** is an interface in the View class that contains a single callback method. This method will; be called by the Android framework when the View to which the listener has been registered is triggered by user interaction with the item in the UI.



5.5.3 Updating a View

To update a View the code must first instatiate an object from the View. The code can then update the object, which updates the screen.

To refer to the View in the code , use the **findViewById()** method of the View class, which looks for a View based on the resource id.

6 Activities and Intents

6.1 Activities (High-Level view)

6.1.1 What is an Activity?

An activity is an application component.

Represents one window and typically fills the screen.

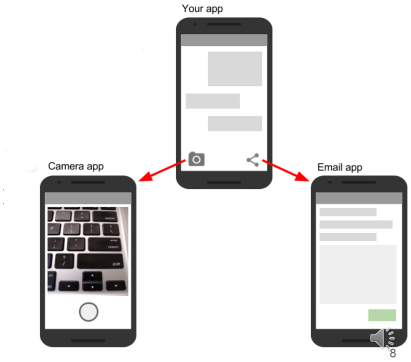
A Java class is typically one Activity in one file

An activity represents a single screen in the app with an interface the user can interact with, for example an email app might have 3 activities : one to shows a list of received emals , one to compose an email and one to read individual messages

6.1.2 What does an activity do?

Apps are often collections of activities that you create yourself or that you reuse from other apps.

Handles user interactions, such as button clicks, text entry and login verification.



6.1.3 Apps and activities

First Activity user sees is typically called **main activity**

Activities are loosely tied together to make uo an app.

Activities can be organized in parent-child relationships in the Android manifest to aid navigation.

An activity has a life cycle : Created , Started , Runs , Paused , Resumed , Stopped and Destroyed.

6.1.4 Layouts and Activities

An Activity typically has a UI layout..

Layout is usually defined in one or more XML files.

Activity inflates layout as part of being created.

6.2 Implementing Activities

6.2.1 Implement new activities

When creating a new project the wizard automatically performs the following steps (1. Define layout in XML, 2. Define Activity Java class , 3. Connect Activity with Layout , 4. Declare Activitu in the Android manifest).

Define Layout in XML

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent">
   <TextView
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="Let's Shop for Food!" />
</RelativeLayout>
```

Define Activity Java class

When creating a new project the MainActivity is , by default , a subclass of the AppCompatActivity class.

This allows to use up-to-date Android app features such as the app bar and Material Design while still enabling the app to be compatible with devices running older versions of Android

The first task in the implementation of an Activity subclass is to implement the standard Activity lifecycle callback methods (such as OnCreate()) to handle the state changes for your Activity.

The one required callback that an app must implement is the onCreate() method.

The system calls this method when it creates the Activity , and all essential components of your Activity should be initialized here.

Connect activity with layout

The onCreate() method setContentView() with the path to a layout file.

The system creates all the initial views from the specified layout and adds them to your Activity. This is often referred to as inflating the layout.

Declare activity in Android manifest

Each Activity in an app must be declared in the AndroidManfest.xml file with the activity element , inside the application section.

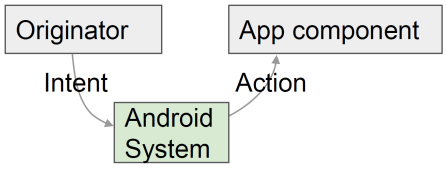
MainActivity needs to include intent-filter to start from launcher. The action element specifies that this is the main entry point to app and only the MainACtivity should include the main action

6.3 Intents

6.3.1 What is an intent?

An Intent is a **description of an operation to be performed**.

An Intent is an object used to request an action from another app component via the Android system.



6.3.2 Starting the main activity

1. When the app is first started from the device home screen
2. The Android runtime sends an Intent to the app to start the app's main activity.

```
<activity android:name=".MainActivity" >

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

</activity>
```

6.4 What can intents do?

- Start an Activity : A button click starts a new Activity for text entry , pass data between one activity and another. Clicking Share opens an app that allows you to post a photo.
- Start a Service : Initiate downloading a file in the background
- Deliver Broadcast : The system infroms everybody that the phone is now charging

6.4.1 Explicit and Implicit intents

Explicit Intent : Starts a specific Activity , for example : Request tea with milk delivered by a specific Cage. Main activity starts with ViewShoppingCart Activity

Impliciti Intent : Asks system to find an Activity that can handle this request, Find an open store that sells green tea. Clicking Share opens a chooser with a list of apps.

6.5 Starting Activities

6.5.1 Start an Activity with an explicit intent

To start a specific Activity , use an explicit Intent.

1. Create an Intent : Intent intent = new Intent(this, ActivityName,class)
2. Use the Intent to start the Activity : startActivity(intent)

6.5.2 Start an Activity with implicit intent

To ask Android to find an Activity to handle your request , use an implicit Intent.

1. Create an Intent : Intent intent = new Intent(action,uri)
2. Use the Intent to start the Activity : startActivity(intent);

Show a web page

```
Uri uri = Uri.parse("http://www.google.com");
Intent it = new Intent(Intent.ACTION_VIEW, uri);
startActivity(it);
```

Dial a phone number

```
Uri uri = Uri.parse("tel:8005551234");
Intent it = new Intent(Intent.ACTION_DIAL, uri);
startActivity(it);
```

6.5.3 How Activities run

All Activity instances are managed by the Android runtime.

Started by an Intent , a message to the Android runtime to run an activity

6.6 Sending and Receiving Data

In addition to open a new activity , with an intent it is possible to pass data from one Activity to another.

In particular , it is possible to use **Intent data** or **Intent extras**.

- Data : One piece of information whose data location can be represented by an URI
- Extras : one or more pieces of information as a collection of key-value pairs.

6.6.1 Intent Data

The Intent data can hold only one piece of information : a URI representing the location of the data you want to operate on :

A Uniform Resource Identifier (URI) is a compact string of characters for identifying an abstract or physical resource

The URI could be:

- a web page URL(http://)
- a telephone number(tel://)
- a geographic location(geo://)

Use the Intent data field when you only have one piece of information that you need to send to the started Activity , that information is a data location that can be represented by a URI.

6.6.2 Extras Data

Intent extras are for any other arbitrary data you want to pass to the started Activity,

Intent extras are stored in a Bundle objects as key and value pairs.

A **Bundle** is a map , in which a key is a string and a value can be any primitive or object type.

To put data into the Intent extras you can use any of the Intent class putExtra() methods or create your own Bundle and put the Bundle into the Intent with putExtras()

Use the Intent extras if you want to pass more than one piece of information to the started Activity and if any of the information you want to pass is not expressible by a URI.

6.6.3 Passing data from one Activity ti another

For **sending data** to an Activity :

1. Create the Intent object
2. Put data or extras into that Intent
3. Start the new Activity

For **receiving data** from an Activity:

1. Get the Intent object, the Activity was started with
2. Retrieve the data or extras from the Intent object

6.6.4 Step 1 : Crete the Intent object

Intent messageIntent = new Intent(this,ShowMessageActivity.class);

6.6.5 Step 2 : Put data into that Intent

Use the setData() method with a URI object to add it to the Intent.

Some examples of using setData() with URIs:

```
// A web page URL
intent.setData(Uri.parse("http://www.google.com"));

// a Sample file URI
intent.setData(Uri.fromFile(new File("/sdcard/sample.jpg")));
```

6.6.6 Step 3 : Start activity

Keep in mind that the data field can only contain a single URI , if you call setData() multiple times only the last value is used and you should use Intent extrax to include additional information.

After you have added the data , you can start the new Activity with the Intent : startActivity(messageIntent);

6.6.7 Sending Extras (with multiple putExtra)

1. Create the Intent object (as seen for data)
2. Put extras into that Intent
3. startActivity(messageIntent)

Use a putExtra() method with a key to put data into the Intent extras. The Intent class defines many putExtra() methods for different kinds of data : for example

Some examples of using setData() with URIs:

```
// A web page URL
intent.setData(Uri.parse("http://www.google.com"));

// a Sample file URI
intent.setData(Uri.fromFile(new File("/sdcard/sample.jpg")));
```

```
public static final String EXTRA_MESSAGE_KEY =
    "com.example.android.twoactivities.extra.MESSAGE";
```

Conventionally you define Intent extra keys as static variables with names that begin with EXTRA_. To guarantee that the key is unique, the string value for the key itself should be prefixed with your app's fully qualified class name.

```
Intent intent = new Intent(this, SecondActivity.class);
String message = "Hello Activity!";
intent.putExtra(EXTRA_MESSAGE_KEY, message);
startActivity(intent);
```

6.6.8 Sending Extras(with a Bundle)

Step 2 (alternative) : if lots of data.

First create a bundle and pass the bundle.

```
Bundle extras = new Bundle();
extras.putString(EXTRA_MESSAGE, "this is my message");
extras.putInt(EXTRA_POSITION_X, 100);
extras.putInt(EXTRA_POSITION_Y, 500);
```

After you've populated the Bundle, add it to the Intent with the putExtras() method (note the "s" in Extras):

```
messageIntent.putExtras(extras);
startActivity(intent);
```

6.6.9 Get data from intents

When you start an Activity with an Intent , the started Activity has access to the Intent and the data it contains.

To retrieve the Intent the Activity (or other component) was started with m use the getIntent() method : Intent intent = getIntent();

Data: Use getData() to get the URI from that Intent:

- `getData();`
  ⇒ `Uri locationUri = intent.getData();`

Extras: Use one of the getExtra() methods to extract extra data out of the Intent object:

- `int getIntExtra (String name, int defaultValue)`
  ⇒ `int level = intent.getIntExtra("level", 0);`
- `Bundle bundle = intent.getExtras();`
  ⇒ Get all the data at once as a bundle.

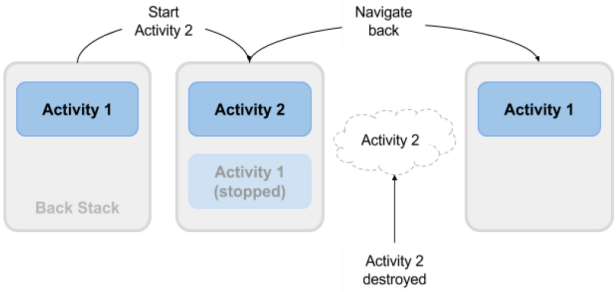
7 Activity Lifecycle and State

7.1 What is the Activity Lifecycle?

The Activity Lifecycle is the set of states an Activity can assume in during its lifetime , from the time it is created , to when it is destroyed.

More formally , a directed graph where , **Nodes** : are all the states an Activity assumes and **Edges** are the callbacks associated with transitioning from each state to the next one.

As the user interacts with the app or other apps on the device , activities move into different states.



When the app starts , the app's main activity is started , comes to foreground , and receives the user focus.

When a second activity starts , the new activity is created and started , and the main activity is stopped.

When the user finishes to interact with the Activity 2 and navigate back , Activity 1 resumes. Activity 2 stops and is no longer needed.

7.2 Activity lifecycle callbacks

7.2.1 Callbacks and when they are called

When an Activity transitions into and out of the different lifecycle states as it runs , the Android system calls several lifecycle callback methods at each stage.

```
onCreate(Bundle savedInstanceState) – static initialization
      onStart() – when Activity (screen) is becoming visible
      onRestart() – called if Activity was stopped (calls onStart())
          onResume() – start to interact with user
          onPause() – about to resume PREVIOUS Activity
      onStop() – no longer visible, but still exists and all state info preserved
  onDestroy() – final call before Android system destroys Activity
```

The activity can be visible or not depending on the current state : Created (not visible yet) , Started (visible), Resume(visible) , Paused (Partially invisible), Stopped (hidden) , Destroyed (gone from memory).

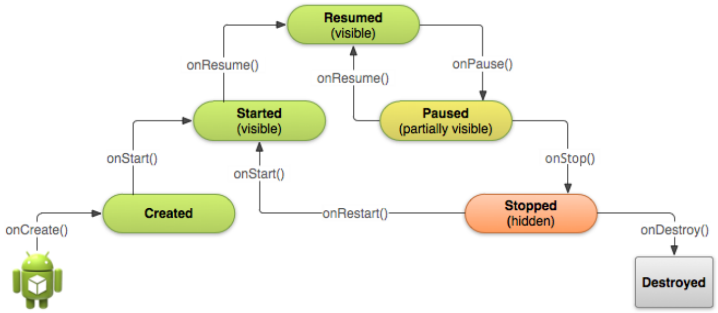
State changes are triggered by user actions , configuration changes and system actions.

7.2.2 Activity states and lifecycle callback methods

All of the callback methods are hooks that can be overridden in each Activity class to define how that Activity behaves when the user leaves and re-enters the Activity.

Keep in mind that the lifecycle states are per Activity , not per app , and you may implement different behaviours at different points in the lifecycle of each Activity.

The figures below shows each of the ACtivity states and the callback methods that occur as the Activity transitions between different states.



7.2.3 Implementing and overriding callbacks

In general it is not needed to implement all the lifecycle callback methods.

Only onCreate() is required.

Override the other callbacks to change default behavior.

However , it is important to understand each one and implement those that ensure your app behaves the way users expect.

Managing the lifecycle of an Activity by implementing callback methods is crucial to developing a strong and flexible app.

Examples : avoiding crashes , consuming system resources not required in specific states, save user progress in the app usage , saving the app state when the screen rotates.

7.3 (1) OnCreate() : Created

Called when the Activity is first created. For example when user taps launcher icon.

Similar to the main() method in other programs.

Does all static setup , perform basic app startup logic such as :

- Setting up the user interface
- Assigning class-scope variables
- Setting up background tasks

Only called once during an activity's lifetime.

Takes a Bundle with Activity's previously frozen state , if there was one.

Created stated is always followed by onStart().

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // The activity is being created.

    // 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);
}
```

7.4 (2) onStart() : Started

Called when the Activity is becoming visible to user.

Can be called more than once during lifecycle.

While onCreate() is called only once when the Activity is created, the onStart() method may be called many times during the lifecycle of the Activity as the user navigates around the app.

Started . like created , is a transient state. After starting , the Activity moves into the resumed (running) state.

Typically you implement onStart() in an Activity as counterpart to the onStop() method.

For example : if you release hardware resources (such as GPS or sensors) when the Activity is stopped. You can re-register those resources in the onStart() method.

```
@Override
protected void onStart() {
    super.onStart();
    // The activity is about to become visible.
}
```

7.5 (3) onRestart() : Started

Called after Activity has been stopped, immediately before it is started again.

Always followed by onStart()

```
@Override
protected void onRestart() {
    super.onRestart();
    // The activity is between stopped and started.
}
```

7.6 (4) onResume() : Resumed/Running

An Activity is in the resumed state when it is initialized , visible on screen and ready to use.

The resumed state is often called in the running state , because it is in the state the user is actually interacting with the app.

- Called when Activity will start interacting with user
- Activity has moved on top of the Activity stack
- Starts accepting user input
- Running state
- Always followed by onPause()

```
@Override
protected void onResume() {
    super.onResume();
    // The activity has become visible
    // it is now "resumed"
}
```

7.7 (5) onPause: Paused

Called when system is about leaving the Activity.

Typically used to : commit unsaved changes to persistend data , stop animations and anything that consumes resources.

Implementations must be fast because the next Activity is not resumed until this method returns.

Followed by either onResume() if the Activity returns back to the front , or onStop() if it becomes invisible to the user.

```
@Override
protected void onPause() {
    super.onPause();
    // Another activity is taking focus
    // this activity is about to be "paused"
}
```

7.8 (6) onStop() : Stopped

Called when the Activity is no longer visible to the user.

New Activity is being started and an existing one is brought in front of this one. This one is being destroyed.

Operations that were too heacy-weight for onPause().

Followed by either onRestart() if Activity is coming back to interact with user , or onDestroy() if Activity is going away

```
@Override
protected void onStop() {
    super.onStop();
    // The activity is no longer visible
    // it is now "stopped"
}
```

7.9 (7) onDestroy : Destroyed

Final call before Activity is destroyed.

User navigates back to previous Activity or configuration changes.

Activity is finishing or system is destroying it to save space.

System may destroy Activity without calling this , so use onPause() or on-Stop() to save data or state.

```
@Override
protected void onDestroy() {
    super.onDestroy();
    // The activity is about to be destroyed.
}
```

8 Activity Instance State

Configuration changes invalidate the current layout or other resources in your activity when the user :

- Rotates the device
- Chooses different system language , so local cahnges
- Enters multi-window mode

8.1 What happens on config change?

On confifuration change , Android : 1. shuts down Activity by calling OnPause , OnStop , OnDestroy.

2. Starts Activity over again by calling : onCreate,onStart,onResume.

8.2 Activity instance state

State information is created while the Activity is running such as : a counter , user text , animation progression.

State is lost when device is rotated, language changes , back-button is pressed , or the system clears memory.

8.2.1 What the system saves

System saves only : Staet of views with unique ID (android:id) such as text entered into EditText. Intent that started activity and data in its extras.

You are responsible for saving other activity and user progress data

8.2.2 Saving instance state

Implement onSavenInstaceState() in your Activity , called by Android runtime when there is a possibility the Activity may be destroyed. Saves data only for this instance of the Activity during current session.

8.2.3 onSaveInstanceState(Bundle outState)

```
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    // save info
    outState.putString("count",
                String.valueOf(mShowCount.getText()));
}
```

8.2.4 Restore instance state

Two ways to retrieve the saved Bundle :

- in onCreate(Bundle mySavedState) : Preferred , to ensure that your user interface , including any saved state, is back up and running as quickly as possible
- Implement callback (called after onStart()) : onRestoreInstanceState(Bundle mySavedState)

8.2.5 Restoring in onCreate()

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

   if (savedInstanceState != null) {
       username = savedInstanceState.getString(“user");
   }
}
```

8.2.6 onRestoreInstanceState(Bundle state)

```
@Override
protected void onRestoreInstanceState (Bundle mySavedState) {
   super.onRestoreInstanceState(mySavedState);

   if (savedInstanceState != null) {
       username = savedInstanceState.getString(“user");
   }
}
```

8.2.7 Instance state and app restart

When you stop and restart a new app session , the Activity instance states are lost and your activities will revert to their default appearance.

If you need to save user data between app sessions, use shared preferences or a database.

9 Activities and Intents Part 2

How to get data back from an Activity

9.1 Returning data to the starting activity

When starting an Activity with an Intent : the originating Activity is paused and the new Activity remains on the screen until the user clicks the back button or the finish() method is called.

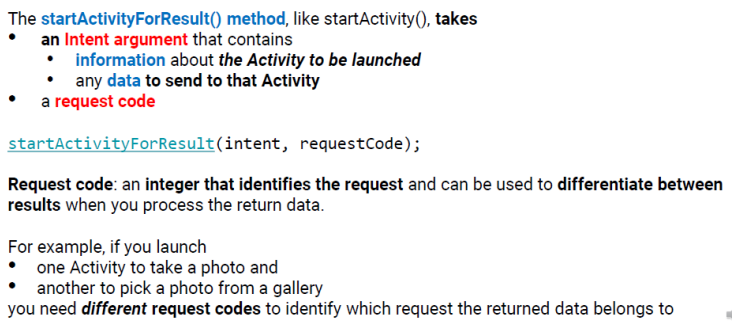
Sometimes when starting an Activity with an Intent , you would like to also get data back from that Intent.

Eg. Start an activity that lets the user pick a prefix, the original Activity needs to receive information about the prefix the user chose back from the launched/new Activity

To launch a new Activity and get a result back , do the following steps :

1. Use startActivityForResult() to start the second Activity
2. To return data from the second Activity : Create a new Intent , put the response data in the Intent using putExtra() , set the resutl to Activity.RESULT_OK or RESULT_CANCELED , if the user cancelled out and call finish() to close the Activity
3. Impement onActivityResult() in first Activity

9.1.1 (1) startActivityForResult()



EXAMPLE :
In the activity that starts the new one :
public static final int CHOOSE_FOOD_REQUEST = 1;
Intent intent = new Intent(this,ChooseFoodItemsActivity.class);
startActivityForResult(inten,CHOOSE_FOO_REQUEST);

9.1.2 (2) Return data and finish second activity

In the new activity to return info :

Intent replyIntent = new Intent(); // No target Activity , Android system directs the response back to the originating Activity automatically.

replyIntent.putExtra(EXTRA_REPLY,reply); // public final static String EXTRA_REPLY = "com.example.myapp.RETURN_MESSAGE";

setResult(RESULT_OK, replyIntent);

finish();

9.1.3 (3) Implement onActivityResult()

In the original activity receive the data :

```
public void onActivityResult(int requestCode, int resultCode, Intent data) {
  super.onActivityResult(requestCode, resultCode, data);

  if (requestCode == CHOOSE_FOOD_REQUEST) { // Identify activity
    if (resultCode == RESULT_OK) { // Activity succeeded
      String reply = data.getStringExtra(SecondActivity.EXTRA_REPLY);
      // … do something with the data
 }}}
```

10 Navigation and Implicit Intents

10.1 Navigation

The majority of the apps will include more than one Activity.

Consistent navigation is an essential component of the overall user experience.

10.1.1 Two forms of navigation

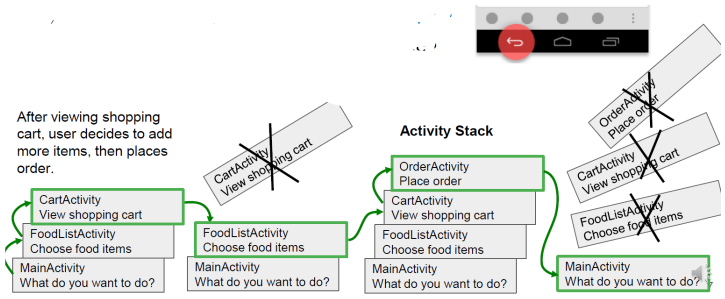
Android system supports two different forms of navigation strategies for your app.

Back (or temporal) navigation

Up (or ancestral) navigation

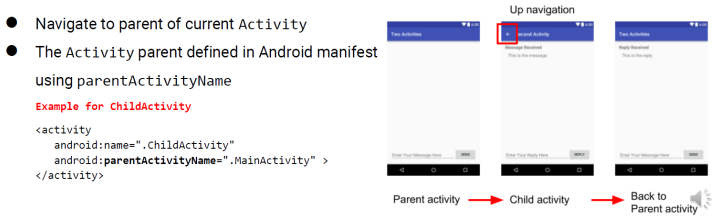
Back Navigation

Allows to return the previous Activity by tapping the device back button controlled by the Android system's back stack and preserves history of recently viewed screens.

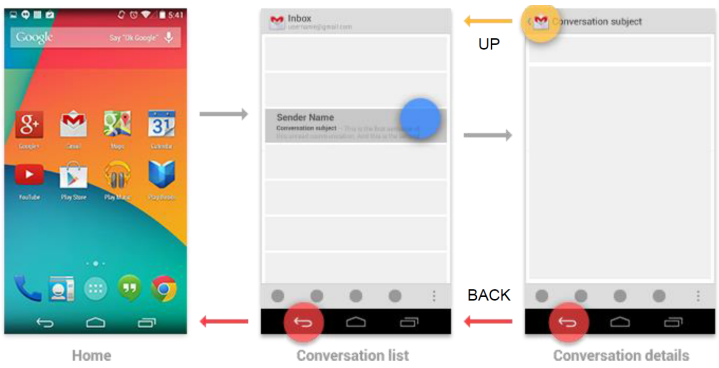


Up Navigation

It is used to navigate within an app based on the explicit hierarchical relationships between screens.



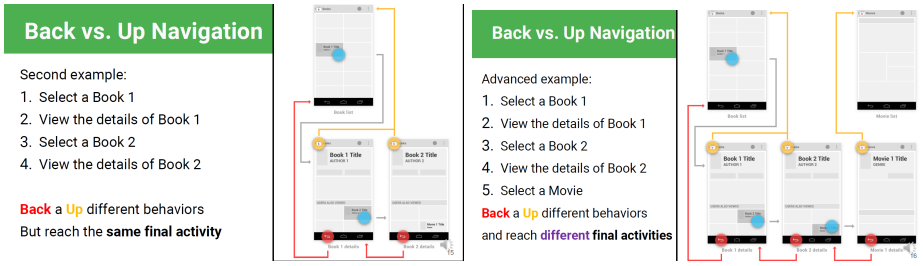
Back vs. Up Navigation



When the previously viewed screen is also the hierarchical parent of the current screen , pressing the Back button has the same result as pressing an Up button (this is a common occurrence).

Up button ensures the user remains within the app.

Back button can return the user to the Home screen or even to a different app.



10.1.2 Back Button

Back button also supports different-behaviors :

- Dismisses floating windows
- Dismisses contextual action bars , and removes the highlight from the selected items
- Hides the onscreen keyboard

10.2 Implicit Intents

10.2.1 What is an Intent?

An Intent is a description of an operation to be performed , Messaging object used to request an action from another app component via the Android system.

Explicit Intent : Starts an Activity of a specific class

Implicit Intent : Asks system to find an Activity class with a registered handler that can handle this request.

10.2.2 Implicit Intent

Start an Activity in another app by describing an action you intend to perform.

Specify an action and optionally provide data with which to perform the action.

Android runtime matches the implicit intent request with registered intent handlers.

If there are multiple matches , an App Chooser will open the let the user decide.

When the Android runtime finds multiple registered activities that can handle an implicit intent , it displays an App Chooser to allow the user to select the handler.

10.2.3 How does implicit Intent work?

The Android Runtime keeps a list of registered Apps.

Apps have to register via AndroidManifest.xml

Runtime receives the request and looks for matches , uses Intent filters for matching from AndroidManifest.xml

If more than one match , shows a list of possible matches and lets the user choose one.

Android runtime starts the request activity.

Show a web page

```
Uri uri = Uri.parse("http://www.google.com");
Intent it = new Intent(Intent.ACTION_VIEW,uri);
startActivity(it);
```

Dial a phone number

```
Uri uri = Uri.parse("tel:8005551234");
Intent it = new Intent(Intent.ACTION_DIAL, uri);
startActivity(it);
```

10.3 Receiving an Implicit Intent

10.3.1 Register your app to receive an Intent

If an Activity in your app have to respond to an implicit Intent (from your own app or other apps) , declare one or more Intent filters in the AndroidManifest.xml file

Each Intent filter specifies the type of Intent it accepts based on the action , data and category for the Intent.

The system will deliver an implicit Intent to your app component only if that Intent can pass through one of your Intent filters.

10.3.2 Intent action,data and category

- **Action** : is the generic action the receiving Activity should perform. The available Intent actions are defined as constants in the Intent class and begin with the word ACTION_
- **Category** : provides additional information about the category of component that should handle the Intent. Intent categories are also defined as constants in the Intent class and begin with the word CATEGORY_
- **Data type** : indicates the MIME type of data the Activity should operate on. Usually , the data type is inferred from th.e URI in the Intent data field, but you can also explicitly define the data type with the setType() method

Intent actions , categories and data types are used both by the Intent object you create in your sending Activity. As well as , in the Intent filters you define in the AndroidManifest.xml file for the receiving Activity.

The Android system uses this information to match an implicit Intent request with an Activity or other component that can handle that Intent.

Intent filter in AndroidManifest.xml

```
<activity android:name="ShareActivity">
  <intent-filter>
    <action android:name="android.intent.action.SEND"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="text/plain"/>
  </intent-filter>
</activity>
```

Intent filters: action and category

- `action` — Match one or more action constants
  - `android.intent.action.VIEW` – matches any Intent with `ACTION_VIEW`
  - `android.intent.action.SEND` – matches any Intent with `ACTION_SEND`

- `category` — additional information (list of categories)
  - `android.intent.category.BROWSABLE`—can be started by web browser
  - `android.intent.category.LAUNCHER`—Show activity as launcher icon

Intent filters: data

- `data` — Filter on data URIs, MIME type
  - `android:scheme="https"`—require URIs to be https protocol
  - `android:host="developer.android.com"`—only accept an Intent from specified hosts
  - `android:mimeType="text/plain"`—limit the acceptable types of documents

An Activity can have multiple filters

```
<activity android:name="ShareActivity">
  <intent-filter>
    <action android:name="android.intent.action.SEND"/>
    ...
  </intent-filter>
  <intent-filter>
    <action android:name="android.intent.action.SEND_MULTIPLE"/>
    ...
  </intent-filter>
</activity>
```

An Activity can have several filters

A filter can have multiple actions & data

```
<intent-filter>
  <action android:name="android.intent.action.SEND"/>
  <action android:name="android.intent.action.SEND_MULTIPLE"/>
  <category android:name="android.intent.category.DEFAULT"/>
  <data android:mimeType="image/*"/>
  <data android:mimeType="video/*"/>
</intent-filter>
```

10.3.3 Receiving an implicit Intent

Once the Activity is successfully launched with an implicit Intent, from that activity it is possible to handle the Intent and its data in the same way you did for an explicit Intent , by :

1. Getting the Intent object with getIntent()
2. Getting Intent data or extras out of that Intent
3. Performing the task the Intent requested
4. Returning data to the calling Activity with another Intent , if needed.

11 User Interaction , Buttons and Clickable Images

In Android app , user interaction typically involves tapping , typing , using gestures but also talking.

The Android framework provides corresponding user interface UI elements such as : buttons , clickable images , menus , keyboards , text entry fields and a microphone.

Android users expects UI elements to act in certain ways , so it is important that your app is consistent with other Android apps.

To satisfy the users , it is important to create a UI that provides predictable choices / behaviors.

11.0.1 User Interaction Design

Important to be obvious , easy and consistent :

- Think about how users will use your app
- Minimize steps
- Use UI elements that are easy to access , understand , use
- Follow Android best practices
- Meet user's expectations

11.1 Buttons

Button are Views that respond to tapping or pressing.

Usually text or visuals indicate what will happen when tapped/pressed.

Buttons can have the following design :

- Text only
- Icon only
- Both text and icon

11.1.1 Button Image

To add an icon to a button : search for the attribute of interest in the attributes list , click "pick a resource" and select the icon.

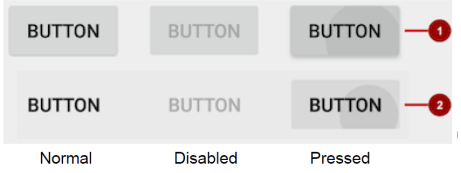
The following attributes manage how an icon is visualize in a button.

android:drawableBottom	The drawable to be drawn below the text.
android:drawableEnd	The drawable to be drawn to the end of the text.
android:drawableLeft	The drawable to be drawn to the left of the text.
android:drawablePadding	The padding between the drawables and the text.
android:drawableRight	The drawable to be drawn to the right of the text.
android:drawableStart	The drawable to be drawn to the start of the text.
android:drawableTint	Tint to apply to the compound (left, top, etc.) drawables.
android:drawableTintMode	Blending mode used to apply the compound (left, top, etc.) drawables tint.
android:drawableTop	The drawable to be drawn above the text.

11.1.2 Raised and Flat Buttons

Android offers several types of Button elements , including raised buttons and flat buttons.

Each button has three states : Normal , Disabled and Pressed.



A **raised button** is an outline rectangle or rounded rectangle that appears lifted from the screen - the shading around it indicates that it is possible to tap or click it. The raised button can show a text , an icon , or both)default style)

A **flat button** , also known as a text button or borderless button , is a text-only that looks flat and doesn't have a shadow. The major benefit of flat buttons is simplicity : a flat button does not distract the user from the main content as much as a raised button does.

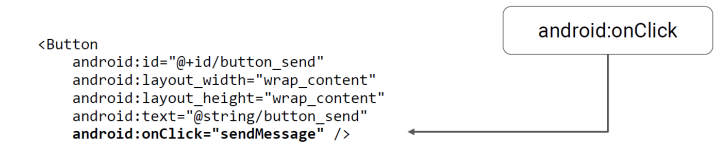
Flat buttons are useful for dialogs that require user interaction. In this case , the button uses the same font and style as the surrounding text to keep the look and feel consistent across all the elements in the dialog.

To create a flat button add the following attribute to your button : style=”? android:attr borderlessButtonStyle

11.1.3 Responding to button taps

In XML : Android Studio provides a shortcut for setting up an OnClickListener for the clickable object in your Activity code , and for assigning a callback method: use the android:OnClick attribute within the clickable object's element in the XML layout.

use `android:onClick` attribute in the XML layout:



and then define the method (e.g., sendMessage) in the code

11.1.4 Setting listener with onClick callback

In your code: use `OnClickListener` event listener

```
Button buttonSend = findViewById(R.id.button_send);

buttonSend.setOnClickListener(new View.OnClickListener() {
       @Override
       public void onClick(View v) {
         // Do something in response to button click
    }
});
```

11.1.5 How and Where to define Listeners

Case 1

all the Listeners must be initialized as soon as possible, before user gets to interact with the Activity.

Case 1:

Defining the listeners in the onCreate() method

Simple approach

Clutters onCreate when having a lot of Listeners

```java
public class AwesomeButtonActivity extends AppCompatActivity {

    private Button awesomeButton;

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        awesomeButton = findViewById(R.id.awesomeBtn);

        awesomeButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                awesomeButtonClicked();
            }
        });
    }

    private void awesomeButtonClicked() {
        awesomeButton.setText("AWESOME!");
    }
}
```

Case 2

Case 2:

Similar to Case 1 except we assign the implementation to a field in the class.

Easy to refactor from Case 1

Promote reusability of the code

Several other ways available, see for instance:

https://medium.com/@CodyEngel/4-ways-to-implement-onclicklistener-on-android-9b956cbd2928

```java
public class AwesomeButtonActivity extends AppCompatActivity {

    private Button awesomeButton;

    private View.OnClickListener awesomeOnClickListener = new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            awesomeButtonClicked();
        }
    };

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        awesomeButton = findViewById(R.id.awesomeBtn);

        awesomeButton.setOnClickListener(awesomeOnClickListener);
    }

    private void awesomeButtonClicked() {
        awesomeButton.setText("AWESOME!");
    }
}
```

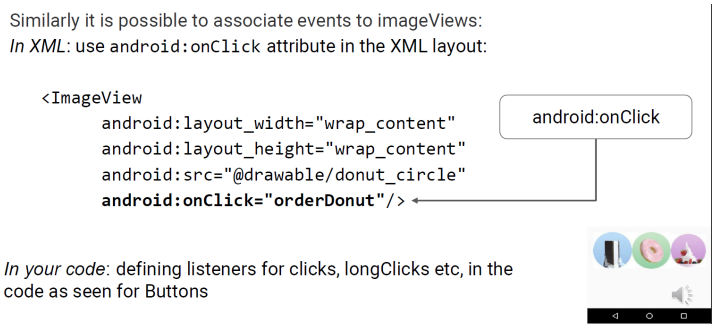
11.1.6 Responding to button LONG taps

Similarly the buttons can respond to other events. For instance to long click

```java
Button button = findViewById(R.id.button);
button.setOnLongClickListener(new View.OnLongClickListener() {
  @Override
  public boolean onLongClick(View v) {
     // Do something in response to button click

     //return true if the callback consumed the long click, false otherwise.
     //return boolean true at the end of OnLongClickListener to indicate
       you don't want further processing
     return true;
     //return false;
  }
});
```

https://stackoverflow.com/questions/5428077/android-why-does-long-click-also-trigger-a-normal-click

11.1.7 Responding to ImageView taps



12 Data Storage

12.1 Storing Data

Android provides several options for saving persistent app data.
Possible solutions include :

- Internal Storage : Private data on device memory
- External Storage : App-specific files or Public data on device or external storage
- Shared Preferences : Private primitive data in key-value pairs
- SQLite Databases : Structured data in a private database

The best solution depends on the app specific needs.
Data should be private to the app OR accessible to other apps/user?
How much space the data requires? Complex structure needed?

12.1.1 Storing data beyond Android

- Network Connection : On the web with your own server
- Cloud Backup : Back up app and user data in the cloud
- Firebase Realtime Database : Store and sync data with NoSQL cloud database across clients in realtime

12.2 Files - Internal and External Storage

12.2.1 Android File System

Android uses a file system that is similar to disk-based file systems on other platforms such as Linux

All Android devices have two file storage areas :

- Internal Storage : Private directories for just your app
- External Storage : Public directories

App can browse the directory structure.

12.2.2 Internal Storage

Always available to the app , uses device's file system. Only your app can access files. On app uninstall , system removes all app's files from internal storage.

12.2.3 External Storage

Uses device's file system or physically external storage like SD card.

Not always available , because the SD card can be removed.

World-readable , so any app can read.

On uninstall , system does not remove files.

12.2.4 When to use internal/external storage

Internal is best when you want to be sure that neither the user nor other apps can access your files.

External is best for files that do not require access restrictions , you want to share with other apps and you allow the user to access with a computer.

12.3 Internal Storage

Your app always has permission to read and write files in its internal storage directory.

- Permanent storage directory : getFilesDir()
- Temporary storage directory : getCacheDir() , recommended for small and temporary files totaling less than 1 MB

These locations are encrypted , these characteristics make these locations a good place to store sensitive data that only your app itself can access.

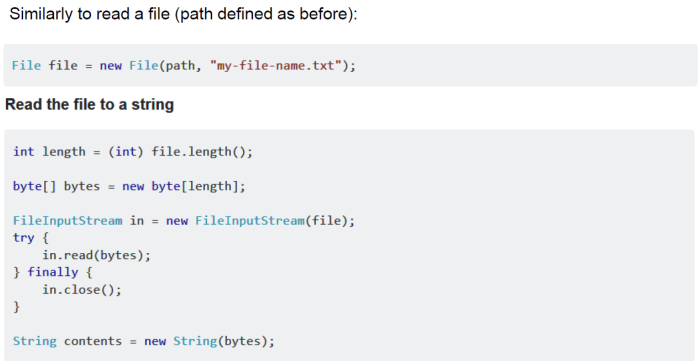
12.3.1 Creating a file

To create a new file in one of these directories , use the **File()** constructor , passing the File provided by one of the methods (getFilesDir() , getCacheDir()), that specifies your internal storage directory. For example : File file = new File(context.getFilesDir() , filename).

Then use standard java.io file operators or streams to interact with files.

12.3.2 Write Text in an Internal File

12.3.3 Read Text in an Internal File



13 Shared Preferences

13.1 What are Shared Preferences?

Read and write small amounts of primitive data as a key/value pairs to a file on the device storage.

The preference file is accessible to all the components of your app , but it is not accessible to other apps.

SharedPreference provides APIs for reading , writing and managing this data.

Save data in onPause() and restore in onCreate()

13.2 Shared Preferences AND Saved Instance State

For both : Data represented by a small number of key/value pairs .

Data is private to the application.

Shared Preferences

Persist data across user sessions , even if app is killed and restarted , or device is rebooted.

Data should be remembered across sessions , such as a user's preferred settings or their game score.

Common use is to store user preferences.

Saved Instance State

Preserve state data across activity instances in same user session.

Data that should not be remembered across sessions , such as the currently selected tab or current state of activity,

Common use is to recreate state after the device has been rotated.

13.3 Creating Shared Preferences

Need only one Shared Preferences file per app. Name it with package name of your app (unique and easy to associate with app).

13.4 getSharedPreferences()

private String sharedPrefile = "com.example.android.hellosharedprefs";

mPreferences = getSharedPreferences(sharedPrefile,MODE_PRIVATE);

MODE argument for getSharedPreferences() is for backwards compatibility.

13.5 Saving Shared Preferences

SharedPreferences.Editor interface , it takes care of all file operations.

Put methods overwrite if key exists.

apply() saves asynchronously and safely.

13.5.1 Saving Shared Preferences (details 1)

Save preferences in the onPause() state of the activity lifecycle using the SharedPreferences.Editor interface

Get a SharedPreferences.Editor , the editor takes care of all the file operations for you.

Add key/value pairs to the editor using the put method appropriate for the data type.

For example putInt() or putString(). These methods will overwrite previously existing values of an existing key.

13.5.2 Saving Shared Preferences (details 2)

Call apply() to write out your changes.

The apply() method saves the preferences asynchronously , off the UI thread.

You do not need to worry about Android component lifecycles and their interaction with apply() writing to disk.

The framework makes sure in-flight disk writes from apply() complete before switching states.

13.6 SharedPreferences.Editor

```
public class MainActivity extends AppCompatActivity {
    private SharedPreferences mPreferences;
    …
    @Override
    protected void onPause() {
       super.onPause();
       SharedPreferences.Editor preferencesEditor = mPreferences.edit();
       preferencesEditor.putInt("count", mCount);
       preferencesEditor.putInt("color", mCurrentColor);
       preferencesEditor.apply();
    }
}
```

13.7 Restore Shared Preferences

Restore in onCreate() in Activity

Get methods take two arguments : the key , the default value (if the key cannot be found).

Use default argument so you do not have to test whether the preference exists in the file.

13.7.1 Getting data in onCreate()

13.8 Clearing

Call clear() on the SharedPreferences.Editor and apply changes.

You can combine calls to put and clear. However , when you apply() , clear() is always done first, regardless of order!

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

   String sharedPrefFile = "com.example.simplesavingsapp";
   mPreferences = getSharedPreferences(sharedPrefFile, MODE_PRIVATE);
   mCount = mPreferences.getInt("count", 1);
   mCurrentColor = mPreferences.getInt("color", 0);
   … // use that values e.g., for initializing UI widgets
}
```

13.8.1 clear()

SharedPreferences.Editor preferencesEditor = mPreferences.edit();
preferencesEditor.clear(); preferencesEditor.apply();

14 Data Storage Part 2

14.1 Android Storage Recap

- Files
  - App-specific storage
    - Internal
    - External
  - Shared storage
    - Media & Doc
- Preferences
- Databases

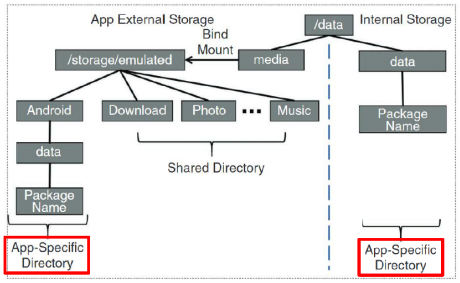


Image adapted from:
IEEE Security & Privacy
Demystifying Android's Scoped Storage Defense. Sept.-Oct. 2021, pp. 16-25, vol. 19
DOI Bookmark: 10.1109/MSEC.2021.3090564

14.2 App-specific files Storage

Both Internal and External storage include a dedicated location for storing persistent files and storing cache data.

Files stored in these directories are meant for use only by your app , otherwise use Shared storage (Photo , Video , Docs , etc).

When storing sensitive data (data that should not be accessible from any other app), use :

- Internal Storage
- Preferences

- Database

Internal storage → data being hidden from users

When the user uninstalls your app , the files saved in app-specific storage are removed.

14.3 External Storage for App-specific files

If internal storage does not provide enough space to store app-specific files → use external storage.

The system provides directories within external storage where an app can organize files that provide value to the user only within your app.

The files in these directories are not guaranteed to be accessible, such as when a removable SD is taken out of the device, If your app's functionality depends on these files → internal storage

14.3.1 Always check availability of storage

Because the external storage may be unavailable , such as when the user has mounted the storage to a PC or has removed the SD card that provides the external storage.

You should always verify that the volume is available before accessing it.

```
/* Checks if external storage is available for read and write */

public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}
```

if the returned state is equal to MEDIA_MOUNTED, then you can read and write your files

```
/* Checks if external storage is available to at least read */
public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state) ||
        Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        return true;
    }
    return false;
}
```

Returns the current state of the primary external storage media…

otherwise use (e.g., for SD card etc):
`String getExternalStorageState (File path)`

14.3.2 Accessing External storage directories

1. Get a path using getExternalFilesDir()
2. Create file

Example :
File path = getExternalFilesDir(Environment.DIRECTORY_PICTURES);
File file = new File(path,"DemoPicture.jpg");

14.4 Select a physical storage location

A device that allocates a partition of its internal memory as external storage can also provide an SD card slot.

This means that the device has multiple physical volumes that could contain external storage, so you need to select which one to use for your app-specific storage.

```
File[] externalStorageVolumes =
            ContextCompat.getExternalFilesDirs(getApplicationContext(), Environment.DIRECTORY_PICTURES);

// probably a partition of the device internal memory as external storage
File pathPrimaryExternalStorage = externalStorageVolumes[0];
// probably this is the SD card
File pathSecondaryExternalStorage = externalStorageVolumes[1];
```

the first element in the returned array (i.e., [0]) is considered the primary external storage volume

14.4.1 How much storage left?

If there is not enough space , throus IOException.

If you know the size of the file , check against space : getFreeSpace() , getTotalSpace().

If you do not know how much space is needed , try/catch IOException.

14.4.2 Delete files no longer needed

External storage : myFile.delete();

Internal storage : myContext.deleteFile(fileName);

14.4.3 Do not delete the user's files!

When the user uninstalls your app , your app's private storage directory and all its contents are deleted.

Do not use private storage for content that belongs to the user!

For example : photos captured or edited with you app , music the user has purchased with your app.

15 AsyncTask

15.1 Threads

15.1.1 The Main thread

When an Android app starts , it creates the main thread, which is often called **UI thread**.

The UI thread needs to give its attention to drawing the UI and keeping the app responsive to user input.

If the UI waits too long for an operation to finish , it becomes unresponsive and so the user is not happy.

The main thread must be fast , otherwise the app will be blocked.

15.1.2 What is a long running task?

Examples of possible long running task :

- Network operations
- Long calculations
- Loading data
- Interacting with Databases

15.1.3 Two rules for Android threads

Do not block the UI thread. Compete each task in less than 16 ms for each screen.

Do not access the Android UI toolkit from outside the UI thread.

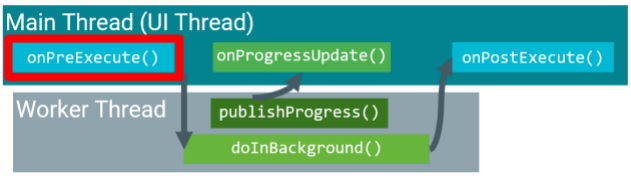
15.2 AsyncTask

AsyncTask allows to perform background operations on a worker thread and publish the result on the UI thread , without needing to directly manipulate threads or handlers.

A worker thread is any thread which is not the main or UI thread.

15.2.1 AsyncTask Execution Steps

When AsyncTask is executed , it goes through several steps :

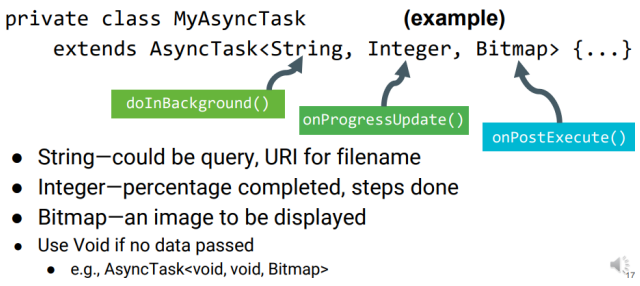


- **onPreExecuted()** : is invoked on the UI thread before the task is executed (normally used to set up the task)
- **doInBackground()** : is invoked on the background thread immediately after onPreExecute() finishes. Performs a backgreound computation , returns a result , and passes the result ot onPostExecute(). This method can also call **publishProgress(Progress...)** to publish one or more units of progress
- **onProgressUpdate()**: Runs on the main thread , receives calls from publishProgress() from background thread.
- **onPostExecute()**: runs on the UI thread after the background computation has finished.

15.2.2 Creating an AsyncTask

Subclass AsyncTask : private class MyAsyncTask extends AsyncTask < type1,type2,type3 > {...}

1. Params - Provide data type (type1) sent to doInBackground()
2. Progress - Provide data type (type2) of progress units for onProgressUpdate()
3. Result - Provide data type (type3) of result for onPostExecute()



15.2.3 AsyncTask Example

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
            …
    }

    public void startTask (View view) {
        // Start the AsyncTask.
        // The AsyncTask has a callback that will update the text view.
        new DownloadFilesTask().execute(url1, url2, url3);
    }

    private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {

        protected Long doInBackground(URL... urls) {
            int count = urls.length;
            long totalSize = 0;
            for (int i = 0; i < count; i++) {
                totalSize += Downloader.downloadFile(urls[i]);
                publishProgress((int) ((i / (float) count) * 100));
                // Escape early if cancel() is called
                if (isCancelled()) break;
            }
            return totalSize;
        }

        protected void onProgressUpdate(Integer... progress) {
            setProgressPercent(progress[0]);
        }

        protected void onPostExecute(Long result) {
            showDialog("Downloaded " + result + " bytes");
        }
    }
}
```

AsyncTask (template) example

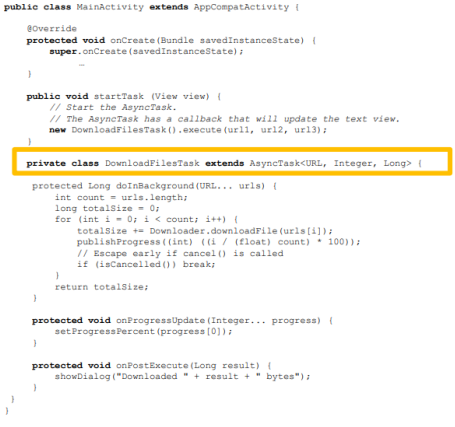
- The "Three Dots" in java is called the **Variable Arguments** or **varargs**
- It allows the method to accept zero or multiple arguments
- Varargs are very helpful if you don't know how many arguments you will have to pass in the method

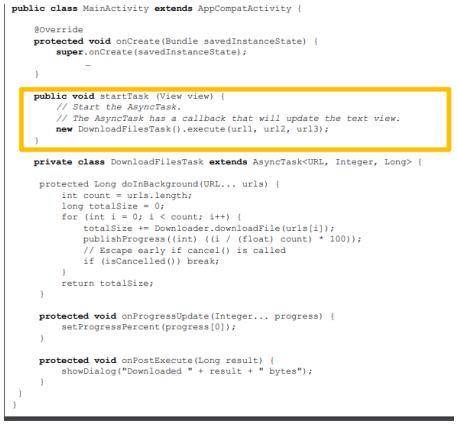
18

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
            …
    }

    public void startTask (View view) {
        // Start the AsyncTask.
        // The AsyncTask has a callback that will update the text view.
        new DownloadFilesTask().execute(url1, url2, url3);
    }

    private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {

        protected Long doInBackground(URL... urls) {
            int count = urls.length;
            long totalSize = 0;
            for (int i = 0; i < count; i++) {
                totalSize += Downloader.downloadFile(urls[i]);
                publishProgress((int) ((i / (float) count) * 100));
                // Escape early if cancel() is called
                if (isCancelled()) break;
            }
            return totalSize;
        }

        protected void onProgressUpdate(Integer... progress) {
            setProgressPercent(progress[0]);
        }

        protected void onPostExecute(Long result) {
            showDialog("Downloaded " + result + " bytes");
        }
    }
}
```

AsyncTask (template) example

19





16 Sensors and Charts

16.1 Sensors in Android

Most Android-powered devices have built-in sensors that measures : motion , orientation , and various environmental conditions.

These sensors are capable of providing raw data with high precision and accuracy.

The Android platform supports three broad categories of sensors:

- Motions sensors : measures acceleration forces and rotational forces along three axis. (accelerometers , gravity sensors , gyroscopes and rotational vectors)
- Environmental sensors : measures various environmental parameters , such as ambient air temperature and pressure , illumination and humidity
- Position sensors: measures the physical condition of the device. Orientation sensors.

16.2 Android Sensor Framework

Android sensor framework allows to access sensors available on the device and acquire raw sensor data.

For example , you can use the sensor framework to do the following:

- Determine which sensors are available on a device
- Determine individual sensor's capabilities , such as its maximum range, power requirements and resolution
- Acquire raw sensor data and define the minimum rate at which you acquire sensor data-
- Register and unregister sensor event listeners that monitor sensors changes.

16.3 Steps for using sensors

A typical application based on sensor APIs requires to perform two basic tasks:

- Identifying sensors and sensor capabilities : useful if the application has features that rely on specific sensor types or capabilities. For example , you may want to identify all the sensors that are present on a device and disable any application features that rely on sensors that are not present
- Monitor sensor events: to acquire raw sensor data. **A sensor event occurs every time a sensor detects a change** in the parameters it is measuring. A sensor event provides you with four pieces of information : Name of the sensor that triggered the event, timestamp for the event , accuracy of the event , raw sensor data that triggered the event.

Sensor	Type	Description	Common Uses
TYPE_ORIENTATION	Software	Measures degrees of rotation that a device makes around all three physical axes (x, y, z). As of API level 3 you can obtain the inclination matrix and rotation matrix for a device by using the gravity sensor and the geomagnetic field sensor in conjunction with the getRotationMatrix() method.	Determining device position.
TYPE_PRESSURE	Hardware	Measures the ambient air pressure in hPa or mbar.	Monitoring air pressure changes.
TYPE_PROXIMITY	Hardware	Measures the proximity of an object in cm relative to the view screen of a device. This sensor is typically used to determine whether a handset is being held up to a person's ear.	Phone position during a call.
TYPE_RELATIVE_HUMIDITY	Hardware	Measures the relative ambient humidity in percent (%).	Monitoring dewpoint, absolute, and relative humidity.
TYPE_ROTATION_VECTOR	Software or Hardware	Measures the orientation of a device by providing the three elements of the device's rotation vector.	Motion detection and rotation detection.
TYPE_TEMPERATURE	Hardware	Measures the temperature of the device in degrees Celsius (°C). This sensor implementation varies across devices and this sensor was replaced with the TYPE_AMBIENT_TEMPERATURE sensor in API Level 14	Monitoring temperatures.

16.4 Identifying the Available Sensors

To identify the sensors that are on a device you first need to get a reference to the sensor service. To do this, you create an instance of the `SensorManager` class by calling the `getSystemService()` method and passing in the `SENSOR_SERVICE` argument. For example:

KOTLIN JAVA

```
private SensorManager sensorManager;
...
sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
```

Next, you can get a listing of every sensor on a device by calling the `getSensorList()` method and using the `TYPE_ALL` constant. For example:

KOTLIN JAVA

```
List<Sensor> deviceSensors = sensorManager.getSensorList(Sensor.TYPE_ALL);
```

16.5 Finding a specific sensor

You can also determine whether a specific type of sensor exists on a device by using the `getDefaultSensor()` method and passing in the type constant for a specific sensor. If a device has more than one sensor of a given type, one of the sensors must be designated as the default sensor. If a default sensor does not exist for a given type of sensor, the method call returns null, which means the device does not have that type of sensor. For example, the following code checks whether there's a magnetometer on a device:

KOTLIN JAVA

```
private SensorManager sensorManager;
...
sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
if (sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD) != null){
    // Success! There's a magnetometer.
} else {
    // Failure! No magnetometer.
}
```

16.6 Capabilities and attributes of sensors

getResolution() : Returns the resolution of the sensor in the sensor's unit.
getMaximumRange(): maximum range of the sensor in the sensor's unit

16.7 Monitoring Sensor Events

To monitor raw sensor data you need to impplement two callback methods that are exposed through the SensorEventListener interface: onSensorChanged() and onAccuracyChanged(). The Android system calls these methods whenever the following events occur:

- A sensor reports a new value: in this case the system invokes the onSensorChanged() method , providing you with a SensorsEvent object. A SensorsEventObject contains information about the new sensor data, including: the accuracy of the data, the sensor that generated the data , the timestamp at which data was generated, and the new data that the sensor recorded.
- A sensor's accuracy changes. In this case the system invokes the onAccuracyChanged() method, providing you with a reference to the sensor object that changed and the new accuracy of the sensor.

onSensorChanged() Example

The following code shows ***how to use the onSensorChanged()*** method to **monitor data from the light sensor**.

```
public class SensorActivity extends Activity implements SensorEventListener {
    private SensorManager sensorManager;
    private Sensor mLight;

    @Override
    public final void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        mLight = sensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
    }

    @Override
    public final void onAccuracyChanged(Sensor sensor, int accuracy) {
        // Do something here if sensor accuracy changes.
    }
```

```
    @Override
    public final void onSensorChanged(SensorEvent event) {
        // The light sensor returns a single value.
        // Many sensors return 3 values, one for each axis.
        float lux = event.values[0];
        // Do something with this sensor value.
    }

    @Override
    protected void onResume() {
        super.onResume();
        sensorManager.registerListener(this, mLight, SensorManager.SENSOR_DELAY_NORMAL);
    }

    @Override
    protected void onPause() {
        super.onPause();
        sensorManager.unregisterListener(this);
    }
}
```

Android Developers - Sensors Overview
https://developer.android.com/guide/topics/sensors/sensors_overview

17 Databases for Mobile Apps

In an Android app it is possible to use two different kinds of DBs

Local : if you want your data to be advice specific (stay local) , then you should go with a local DB. A local DB might be preferable choice over remote DB when you have to simply save data locally and avoid the server request.

Remote : If the app needs to synchronize data across all its users or involve live data being fed to a user's request then you need to use remote database

17.1 Databases for Android (Remote)

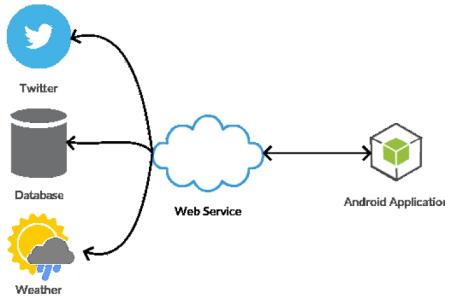
An android application should not directly access to a database deployed on a remote server.

Best practices require to implement a web service between the database and the Android application.

Having a web service layer reduces the complexity of the Android application and reduce the dependency on database specific operations.

The problem of accessing a client/server database like MySQL from Android application can be defined as the problem of consuming a web service hosted somewhere

Do not need to care about what is behind the web service and what you need is the API endpoints exposed in the web service.



17.2 Databases for Android (Local)

On the other hand , if all the information should be stored on a mobile device, there are only some different options including SQLite (stores data to a text file on a device) and Realm

17.2.1 SQLite

The inbuilt SQLite core library is within the Android OS. It will handle CRUD (Create , Read , Update , Delete) operations required for a database.

Java classes and interfaces for SQLite are provided by the android.database.

But this conventional method has its own disadvantages. You have to write long repetitive code , which will be consuming as well as prone to mistakes. It is very difficult to manage SQL queries for a complex relational database

17.2.2 Room Persistence Library

To overcomae this , Google has introced RPL.

This acts as an abstraction layer for the existing SQLite APIs.

All the required packages, parameters , methods , and variables are imported into an Android project by using simple annotations.

It helps to assist developers implementing local SQLite database transactions. It helps to avoid the boilerplate code that was previously associated with interacting with the SQLite database.

Essentially the room persistence library allows developers to easily convert SQLite table data into java objects.