Mobile and Ubiquitous Computing 2022-23

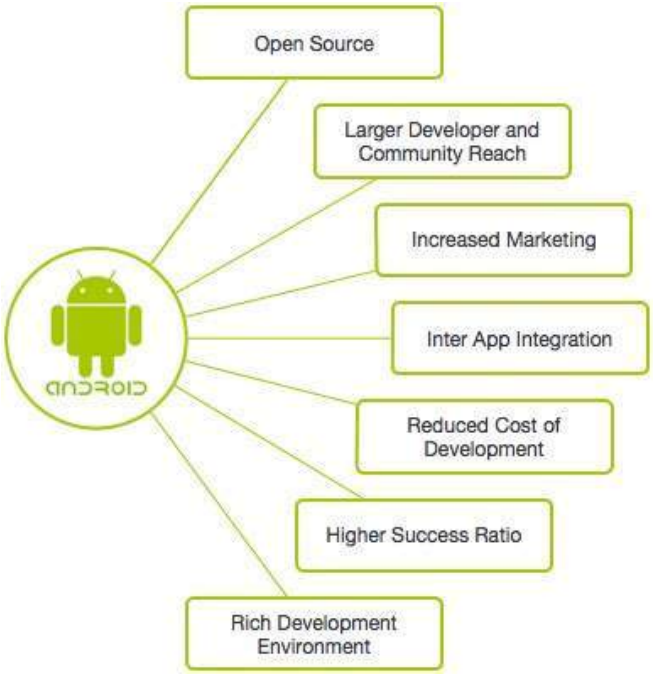
MEIC-A, MEIC-T, METI

Introduction to Android

ANDROID BASICS

The Beginning

- Android was the flagship product of Android Inc.
  - it was a **small startup founded in 1999** to develop a new user-friendly location-aware mobile platform
  - **acquired by Google in 2005**



- The **first Android** phone ran Android version 1.0:
  - it was launched in **September 2008** (HTC Dream)

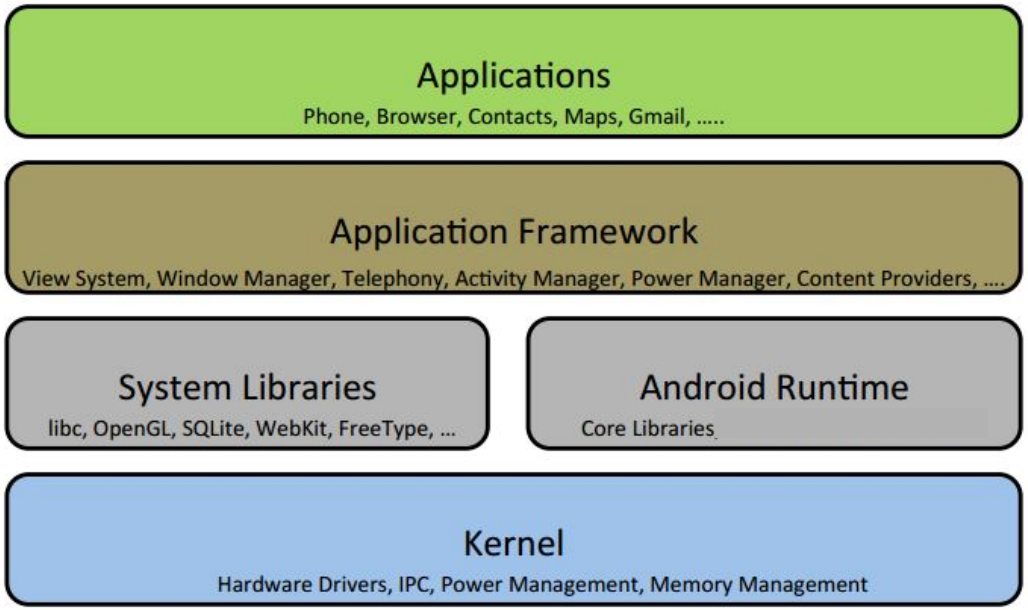
- Android rapid evolution:
  - fast sequence of **updates**
  - growing number of **partners** launching their own mobile devices
  - also used across a wide **variety of mobile devices** (e.g. tablets, mp3 players, netbooks, in-flight entertainment systems
  - Android runs around **2000 million devices** around the world and is still growing at an unparalleled rate

Android Main Features

Feature	Description
Beautiful UI	Android OS basic screen provides a beautiful and intuitive user interface.
Connectivity	GSM/EDGE, IDEN, CDMA, EV-DO, UMTS, Bluetooth, Wi-Fi, LTE, NFC and WiMAX.
Storage	SQLite, a lightweight relational database, is used for data storage purposes.
Media support	H.263, H.264, MPEG-4 SP, AMR, AMR-WB, AAC, HE-AAC, AAC 5.1, MP3, MIDI, Ogg Vorbis, WAV, JPEG, PNG, GIF, and BMP
Messaging	SMS and MMS
Web browser	Based on the open-source WebKit layout engine, coupled with Chrome's V8 JavaScript engine supporting HTML5 and CSS3.
Multi-touch	Android has native support for multi-touch which was initially made available in handsets such as the HTC Hero.

Feature	Description
Multi-tasking	User can jump from one task to another and same time various application can run simultaneously.
Resizable widgets	Widgets are resizable, so users can expand them to show more content or shrink them to save space
Multi-Language	Supports single direction and bi-directional text.
GCM	Google Cloud Messaging (GCM) is a service that lets developers send short message data to their users on Android devices, without needing a proprietary sync solution.
Wi-Fi Direct	A technology that lets apps discover and pair directly, over a high-bandwidth peer-to-peer connection.
Android Beam	A popular NFC-based technology that lets users instantly share, just by touching two NFC-enabled phones together.

Android Overall Architecture (1/2)



The Android Application Framework provides a high level API for application developers to take advantage of the various capabilities of the platform.

Android provides a number of system libraries in C/C++ which are made accessible through the Application Framework.

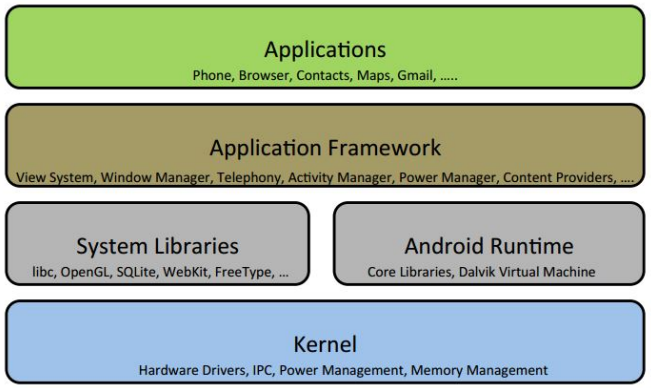
The Android Runtime is the core runtime component of Android

The Android kernel is currently based on version 4.4 of the Linux kernel

- Android is a **Linux-based platform**
- It uses a heavily **customized Linux kernel**
- It is **not another flavor of Linux** because:
  - it **does not support the complete set of standard GNU libraries**, and
  - it uses its own **proprietary windowing system** instead of X-Windows

Android Linux Kernel

- Contains:
  - the hardware abstraction layer (**HAL**)
  - components for **memory management** and **inter-process communication** amongst other low-level functionalities



- It provides:
  - **drivers** for the display, touch input, networking, power management and storage

- One of the main features is:
  - YAFFS2 (Yet Another Flash File System 2) which enables support for **flash-based file systems**.

- Another important feature is the **WakeLock**:
  - it can be used to force a device from going into a low power state
  - it is useful from a user experience perspective since it makes the device more responsive to user interaction
  - it is mostly used by applications and services to request CPU resources
  - it implies that if the OS does not have any active WakeLocks, then the CPU will be shutdown

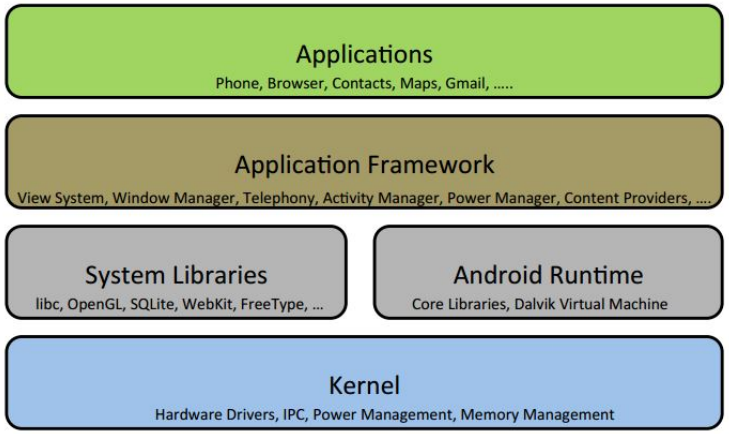
- Binder:
  - proprietary mechanism for **IPC** (RMI included)

Android Runtime

- It is a **process-based VM** optimized for low memory footprint and performance efficiency

- Applications are written in a **dialect of Java**:
  - compiled into bytecodes
  - stored as JVM compatible .class files

- Differences between standard Java and Android APIs:
  - **absence of Abstract Window Toolkit** (AWT), and **Swing** libraries in Android



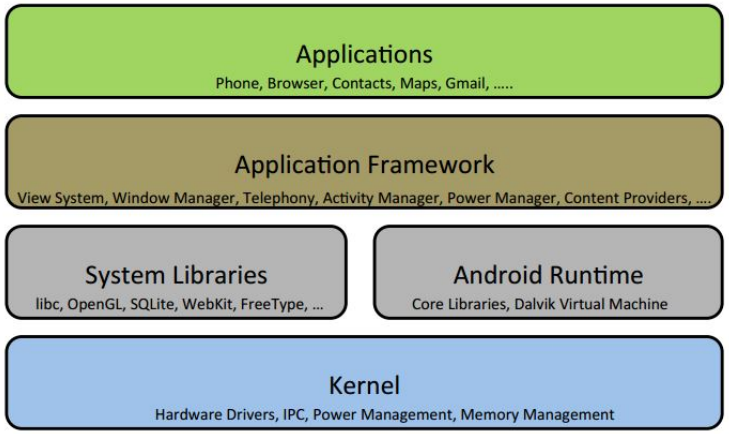
- The JVM class files are converted into **Dalvik Executable** (.dex) files:
  - these are executed by the Android Runtime when the application runs on Android

- Each Android application executes within its own instance of the **Android Runtime**:
  - in turn the Android Runtime runs as a **kernel managed process**

- Ensures that multiple instances of the VM can run at the same time:
  - this **sandboxed** approach ensures security, and if an application requires access to data outside its own sandbox (e.g. such as contacts, text messages, Bluetooth communications)
  - it needs to explicitly request permission during installation on a specific device

System Libraries

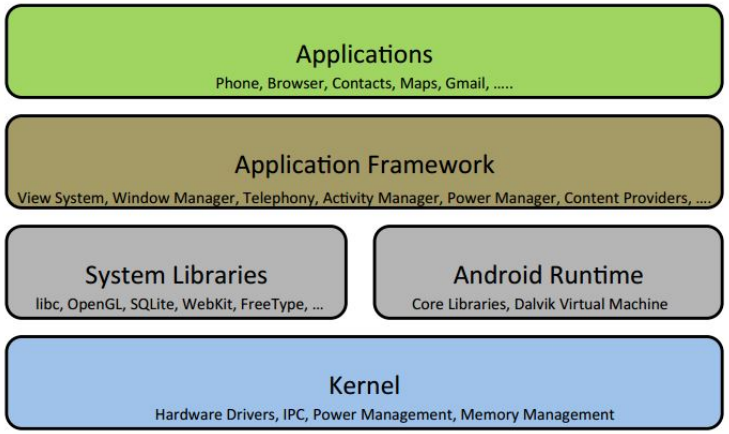
- Android system libraries:
  - in C/C++
  - made accessible through the Application Framework
  - do not provide the complete functionality required of the standard GNU C libraries in Linux



- Some examples:
  - libc typical of C/C++ environments
  - FreeType library provides bitmap and vector-based font rendering
  - SQLite library (database capabilities)
  - OpenGL/ES for 2D and 3D rendering
  - Scalable Graphics Library (2D rendering tasks)
  - compositing of 2D and 3D content is handled by the Surface Manager library
  - LibWebCore library provides a WebKit-based browser engine which can be embedded as a web view within user interfaces of other applications
  - Android Media Library (based on the OpenCORE multimedia framework) handles both images and multimedia content:
    - provides capabilities for recording and playback of commonly used audio, video and still-image formats (e.g. MP3, AAC, H.264, MP4, JPG and PNG)

Application Framework (1/5)

- **Activity Manager –** Controls all aspects of the application lifecycle and activity:
  - interacts with the overall activities running in the system.



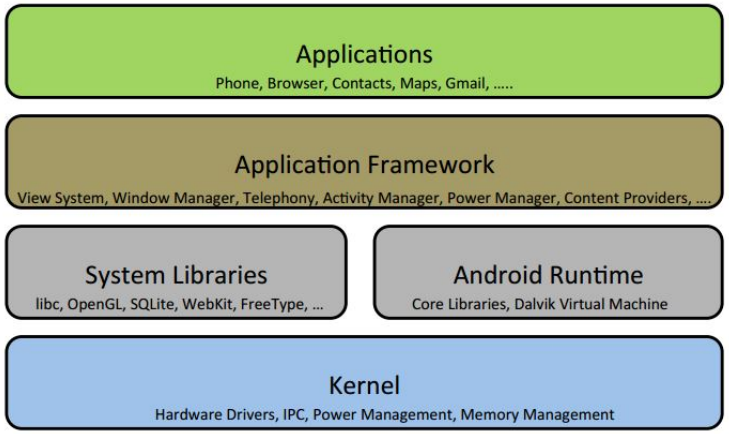
activity: http://developer.android.com/reference/android/app/ActivityManager.html

- **ContentProvider -** allows applications to access and share data with other applications:
  - allows an application to publish its data for access by others
  - the information is provided through a single ContentResolver interface

content providers: http://developer.android.com/guide/topics/providers/content-providers.html

Application Framework (2/5)

- **Resource Manager -** Provides access to non-code embedded resources such as strings, color settings and user interface layouts:
  - resources can be animations, layouts, strings and even image files
  - allow to customize the application based on the type of device it is deployed in or various device settings, such as locale information and language



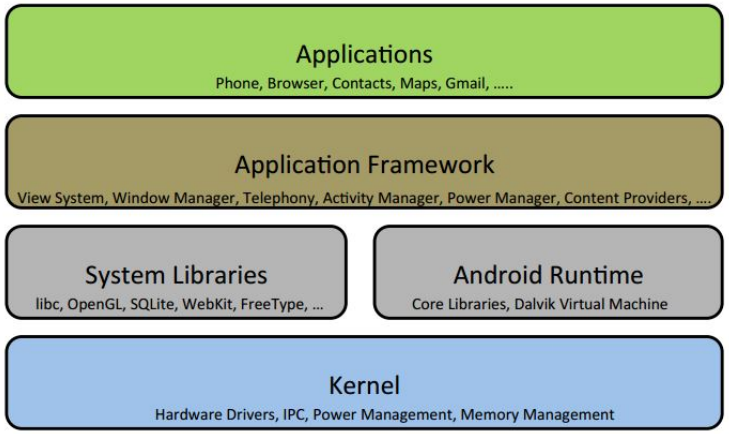
resources: http://developer.android.com/guide/topics/resources/index.html

- **Notifications Manager** − Allows applications to display alerts and notifications to the user:
  - any application can notify the user about specific events when they occur
  - handled by the NotificationManager class, which allows an application to issue notifications (e.g. an icon on the status bar, flashing the device LEDs or the backlight of the device play or by playing a sound or vibrating the device)

notifications: http://developer.android.com/reference/android/app/NotificationManager.html

Application Framework (3/5)

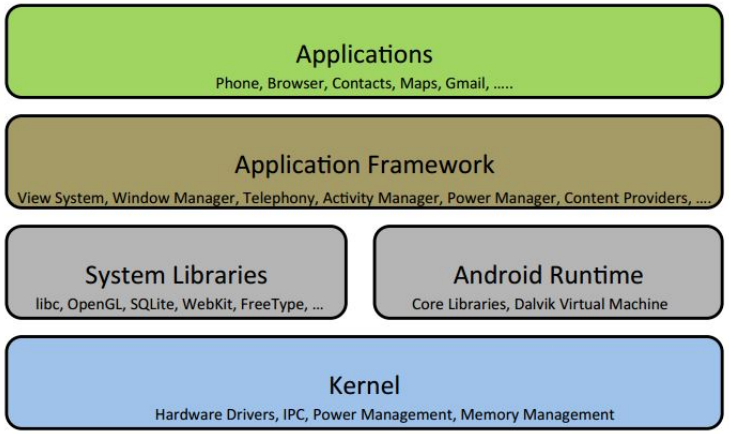
- **View System** − An extensible set of views used to create application user interfaces
- It provides the basic building blocks for creating the user interface of an application:
  - a View object in Android represents a rectangular region of the device's display and is responsible for all the rendering and event handling tasks within that region
  - visual user interface elements in Android are called Widgets and are derived from the View class



user interface: http://developer.android.com/guide/topics/ui/index.html

Application Framework (4/5)

- **LocationManager** - provides access to the localization services available on the device:
  - supports mapping capabilities resulting from the availability of free high-quality turn-by-turn navigation and map applications such as Google Maps
  - third-party developers can utilize this class to create location-based services (LBS)



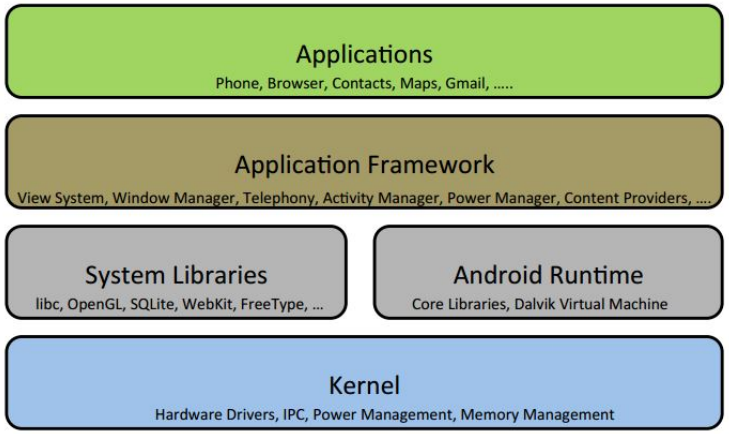
location:http://developer.android.com/reference/android/location/LocationManager.html

- **InputMethodService -** enables to implement custom software keyboards, keypads and even pen input:
  - input is converted into text and passed on to the target UI element

input:http://developer.android.com/reference/android/inputmethodservice/InputMethodService.html

Application Framework (5/5)

- **TelephonyManager** class provides applications with the ability to determine telephony services on the devices and access specific subscriber information
- **SmsManager** allows applications to send data and text messages using the (SMS) protocol



telephony: http://developer.android.com/reference/android/telephony/TelephonyManager.html

- **PowerManager** class provides applications with the ability to control the power state of the device and use a feature called WakeLocks:
  - a WakeLock forces a device to remain on and not go into power-saving mode
  - can provide superior user experience by ensuring the app's interface is immediately responsive to user actions
  - misuse of this capability can lead to extremely poor battery life on the device

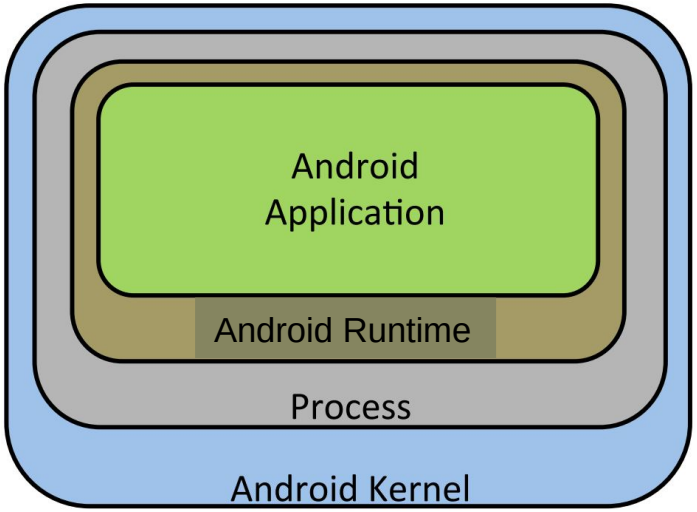
power: http://developer.android.com/reference/android/os/PowerManager.html

ANDROID APPLICATIONS

What Kind of System Entity is an Android Application?

- Android applications written in Java, executed by the Android Runtime (VM)

- Android is not fully compliant with standard Java:
  - there are major differences especially in the user interface libraries



- Each application runs in a sandboxed environment:
  - with its own instance of an Android Runtime which runs within its own kernel managed process

- Android application is installed as a single file of type Android Package (extension: .apk):
  - it contains the compiled code along with data and resource files

- Android package files can be signed or unsigned

What are the Components of an Application?

- Four types of components:

  - Components communicate through Intents

What do the Main Application Components Do?

Activities

- Represent **a screen** with a visual user interface (Activity class)
- Dictate the UI and handle the **user interaction** to the smart phone screen

```
public class MainActivity extends Activity {

}
```

Services:

- Used for **background tasks** such as **time intensive** tasks or inter-application functionalities which do not require direct user interaction (Service class).

```
public class MyService extends Service {

}
```

Content Providers:

- Enable applications to **store and share data** with other applications (ContentProvider class)
- they handle data and database management issues.

```
public class MyContentProvider extends  ContentProvider {

   public void onCreate(){}

}
```

Broadcast Receiver:

- Respond to system-wide broadcast announcements (BroadcastReceiver class)
- Handle communication between Android OS and applications.

```
public class MyReceiver  extends  BroadcastReceiver {

   public void onReceive(context,intent){}

}
```

How do Activities Relate to Applications?

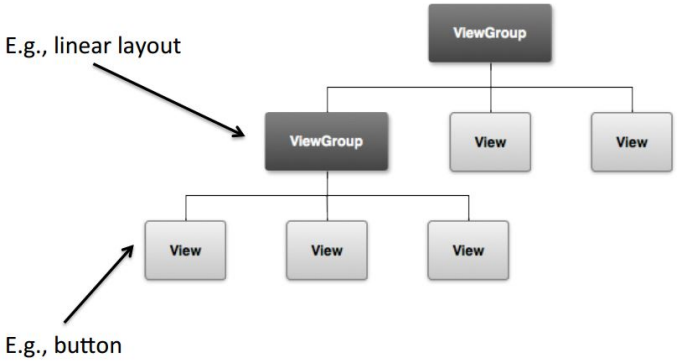
- A typical Android application consists of one or more activities
- Launching an app results in executing one predefined activity (called main activity)

- Facebook App:
  - Activity A allows user to login on Facebook
  - Activity B displays user´s Facebook wall

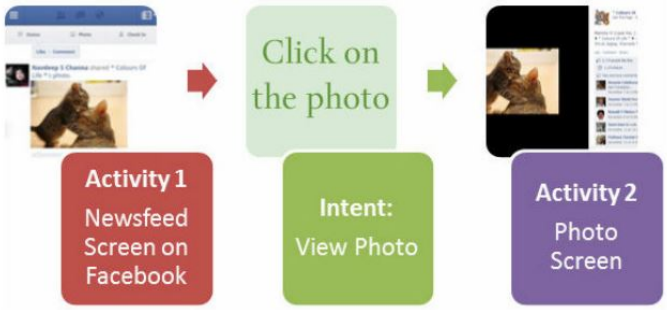
Activities, Views, and Intents: Why are they the base of an application's logic?

- Activities are "similar" to web pages:
  - they provide an interface for users to interact with an app and take actions

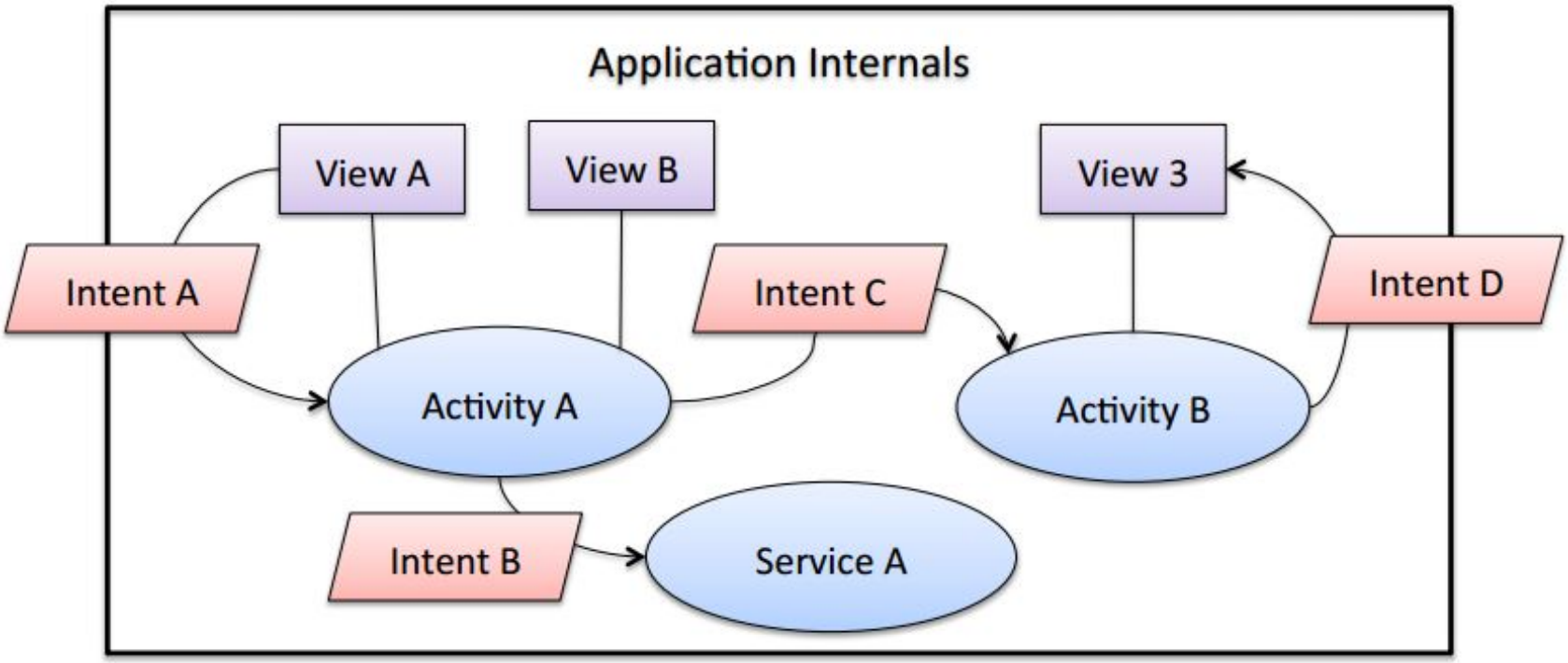
- Views are UI elements hierarchically organized in two types:
  - layouts, and widgets



- Intents are similar to messages that enable communication across components



Activities, Views, and Intents: How do they combine to form an application?



Components
Modules

Views
UI elements (e.g., button)

Intents
Messages

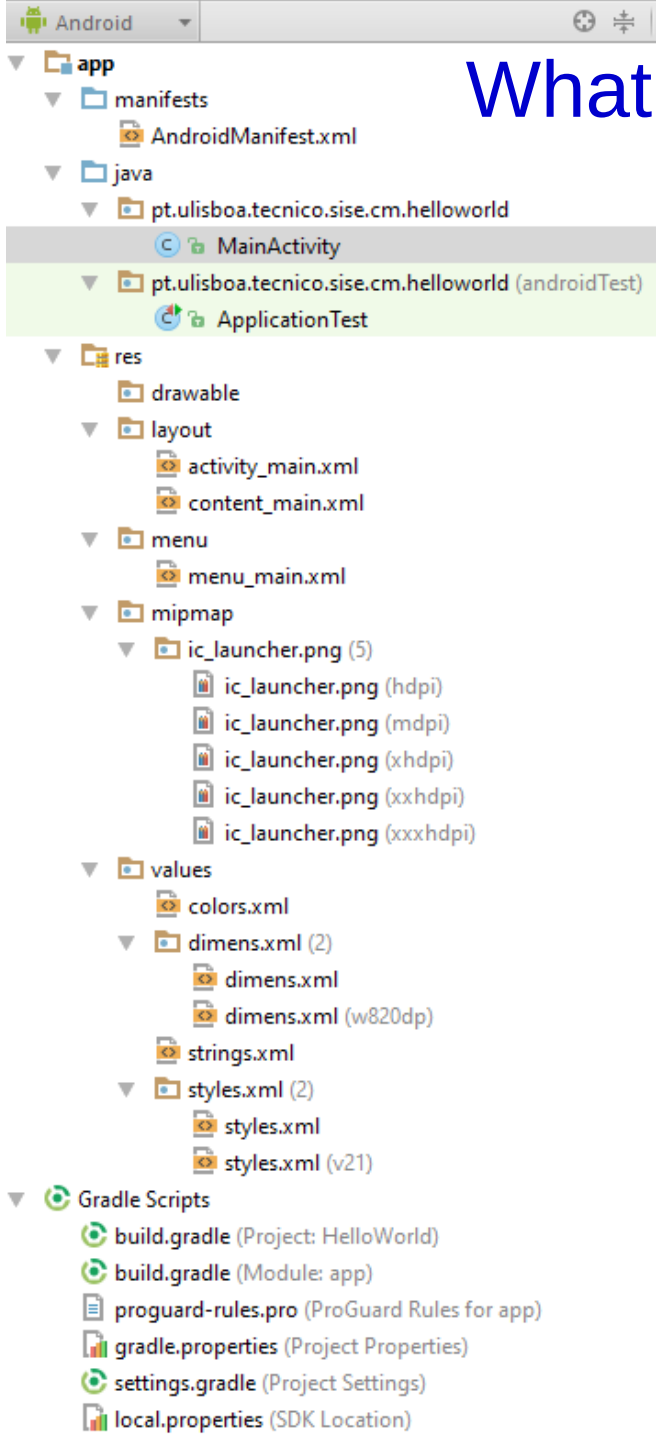
How do Activities Relate to Each Other?

- If an **application** has **multiple** user interface **screens**, then it will contain multiple activity components, e.g., a music player application user interface consists of:
  - one screen for audio playback, and
  - one screen for selecting albums or audio files will have two activity components

- **Any application can** ask the Application Framework to **start** a specific activity in **another application**, e.g. an email application receives a message with an MP3 audio attachment:
  - then, it requests the framework to start the activity representing the music player's audio playback screen
  - so that the user can listen to the contents of the attachment
  - this is done asynchronously through the use of Intents
  - an Intent object provides an abstract description of an action be performed in the form of launching an Activity component or a specific type of component

DEVELOPMENT AND TESTING

What does an Android Project look like?

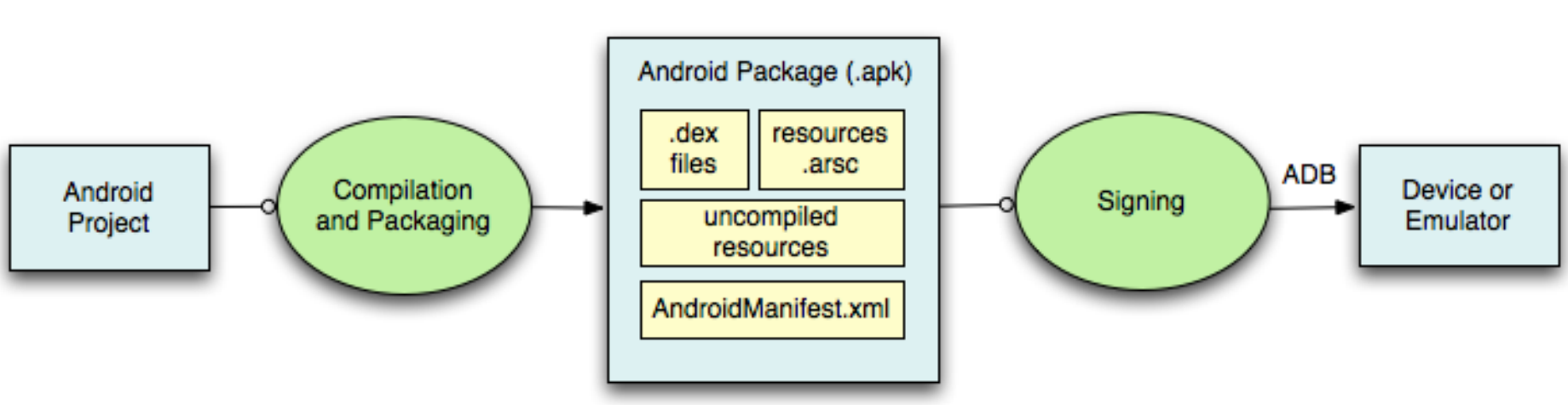


S.N.	Folder, File & Description
1	**src** This contains the **.java** source files for your project. By default, it includes an *MainActivity.java* source file having an activity class that runs when your app is launched using the app icon.
2	**gen** This contains the **.R** file, a compiler-generated file that references all the resources found in your project. You should not modify this file.
3	**bin** This folder contains the Android package files **.apk** built by the ADT during the build process and everything else needed to run an Android application.
4	**res/drawable-hdpi** This is a directory for drawable objects that are designed for high-density screens.
5	**res/layout** This is a directory for files that define your app's user interface.
6	**res/values** This is a directory for other various XML files that contain a collection of resources, such as strings and colours definitions.
7	**AndroidManifest.xml** This is the manifest file which describes the fundamental characteristics of the app and defines each of its components.

What Does the Android Development Process Look Like?

- The Android build process provides:
  - project and module build settings so that
  - your Android modules are compiled and packaged into .apk files, the containers for your application binaries, based on your build settings

- The apk file for each app contains all of the information necessary to run your application on a device or emulator, such as:
  - compiled .dex files (.class files converted to Dalvik byte code),
  - a binary version of the AndroidManifest.xml file,
  - compiled resources (resources.arsc) and uncompiled resource files for your application



What is the Entry Point for an Android App? The Main Activity

- The main activity code is a Java file MainActivity.java:

- this is the actual application file which ultimately gets converted to a DEX executable and runs your application

- R.layout.activity_main refers to the activity_main.xml file located in the res/layout folder

- the onCreate() method is one of many methods that are executed when an activity is loaded.

code generated by the application wizard for Hello World application

```
package pt.ulisboa.tecnico.sise.cm.helloworld;

import android.os.Bundle;
import android.support.design.widget.FloatingActionButton;
import android.support.design.widget.Snackbar;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
import android.util.Log;
import android.view.View;
import android.view.Menu;
import android.view.MenuItem;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);


        Log.d("MainActivity", "This is my first debugging message.");
    }
```

How are App Components Organized? Manifest File

- All components present in an application must be declared:
  - this is done in a manifest file named AndroidManifest.xml

- For each component, the manifest file can also declare an **Intent Filter**:
  - to list its capabilities so that it can respond to intents of specific types

- Apart from this, the manifest file also:
  - declares the **user permissions** required by the application during run time such as access to user data or network access

  - specifies the **hardware and software services** required and the **external libraries** that need to be linked with the application

  - specifies the **API Level** used by the application (given that Android has been evolving rapidly; e.g. with four major versions launched within the last three years)

- API level corresponds to a release version of the Android platform:
  - e.g.: the API level associated with the Ice Cream Sandwich release of Android (version 4.0) is 14

Manifest File (1/3)

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="pt.ulisboa.tecnico.sise.cm.helloworld">

    <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"
            android:label="@string/app_name"
            android:theme="@style/AppTheme.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

- AndroidManifest.xml defines the skeleton of an application
- Whatever component you develop as a part of your application:
  - you must declare all its components in a *manifest.xml*
  - this files resides at the root of the application project directory
  - this file works as an interface between Android OS and your application
  - if you do not declare your component in this file, then it will not be considered by the OS

Manifest File (2/3)

- <application>...</application> enclose the components related to the application:
  - Attribute android:icon points to the application icon available under res/drawable-hdpi
  - the application uses the image named ic_launcher.png located in the drawable folders.

- The <activity> tag is used to specify an activity:
  - android:name attribute specifies the fully qualified class name of the Activity subclass

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="pt.ulisboa.tecnico.sise.cm.helloworld">

    <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"
            android:label="@string/app_name"
            android:theme="@style/AppTheme.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

  - android:label attribute specifies a string to use as the label for the activity

- You can specify multiple activities using <activity> tags

Manifest File (3/3)

- The action for the intent filter is named android.intent.action.MAIN:
  - it indicates that this activity serves as the entry point for the application

- The category for the intent-filter is named android.intent.category.LAUNCHER:
  - it indicates that the application can be launched from the device's launcher icon

- @string/app_name
  refers to the app_name string defined in the strings.xml file, which is "HelloWorld“

- The @string refers to the strings.xml file (see next slide)

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="pt.ulisboa.tecnico.sise.cm.helloworld">

    <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"
            android:label="@string/app_name"
            android:theme="@style/AppTheme.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

- List of tags in the manifest file to specify different Android application components:
  - <activity> elements for activities
  - <service> elements for services
  - <receiver> elements for broadcast receivers
  - <provider> elements for content providers

Strings File

- strings.xml file is located in the res/values folder:
  - it contains all the text that your application uses
  - e.g., the names of buttons, labels, default text, and similar types of strings go into this file
  - this file is responsible for their textual content

```
<resources>
    <string name="app_name">Hello World</string>
    <string name="action_settings">Settings</string>
</resources>
```

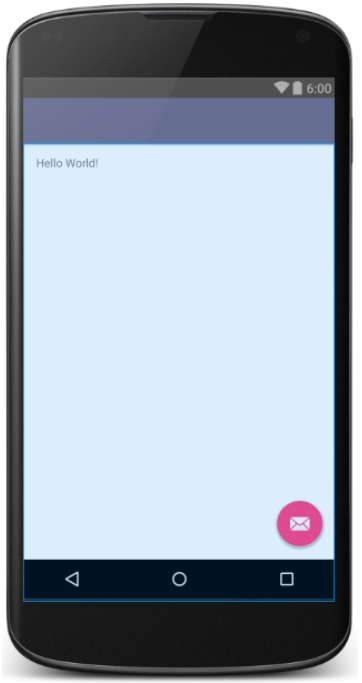
Layout File

- The activity_main.xml / content_main.xml is a layout file available in res/layout directory:
  - it is referenced by your application when building its interface
  - you will modify this file very frequently to change the layout of your application

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:context="pt.ulisboa.tecnico.sise.cm.helloworld.MainActivity"
    tools:showIn="@layout/activity_main">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!" />
</RelativeLayout>
```

"Hello World" application layout file



- The TextView is an Android control used to build the GUI:
  - it has various attributes like android:layout_width, android:layout_height, etc
  - these are being used to set its width and height etc..
- The @string refers to the strings.xml file located in the res/values folder:
  - hence, @string/hello_world refers to the hello string defined in the strings.xml file, which is "Hello World"

Other Resources

- There are many more items which you use to build a good Android application

- Apart from coding for the application, you take care of various other resources like static content that your code uses, such as:
  - bitmaps,
  - colors,
  - layout definitions,
  - user interface strings,
  - animation instructions,
  - etc.

- These resources are always maintained separately in various sub-directories under res/ directory of the project

```
MyProject/
    src/
        MyActivity.java
    res/
        drawable/
            graphic.png
        layout/
            main.xml
            info.xml
        mipmap/
            icon.png
        values/
            strings.xml
```

Resources in Android Studio

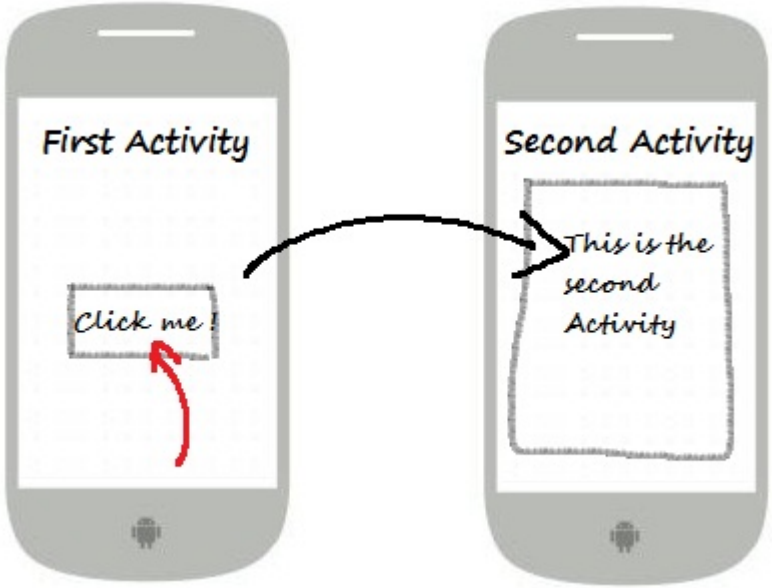
- Resource directories supported inside project res/ directory

Directory	Resource Type
`animator/`	XML files that define property animations.
`anim/`	XML files that define tween animations. (Property animations can also be saved in this directory, but the `animator/` directory is preferred for property animations to distinguish between the two types.)
`color/`	XML files that define a state list of colors. See Color State List Resource
`drawable/`	Bitmap files (`.png` , `.9.png` , `.jpg` , `.gif`) or XML files that are compiled into the following drawable resource subtypes: • Bitmap files • Nine-Patches (re-sizable bitmaps) • State lists • Shapes • Animation drawables • Other drawables See Drawable Resources.
`mipmap/`	Drawable files for different launcher icon densities. For more information on managing launcher icons with `mipmap/` folders, see Managing Projects Overview.
`layout/`	XML files that define a user interface layout. See Layout Resource.
`menu/`	XML files that define application menus, such as an Options Menu, Context Menu, or Sub Menu. See Menu Resource.
`raw/`	Arbitrary files to save in their raw form. To open these resources with a raw `InputStream` , call `Resources.openRawResource()` with the resource ID, which is `R.raw.filename` . However, if you need access to original file names and file hierarchy, you might consider saving some resources in the `assets/` directory (instead of `res/raw/`). Files in `assets/` are not given a resource ID, so you can read them only using `AssetManager` .

Directory	Resource Type
`values/`	XML files that contain simple values, such as strings, integers, and colors. Whereas XML resource files in other `res/` subdirectories define a single resource based on the XML filename, files in the `values/` directory describe multiple resources. For a file in this directory, each child of the `<resources>` element defines a single resource. For example, a `<string>` element creates an `R.string` resource and a `<color>` element creates an `R.color` resource. Because each resource is defined with its own XML element, you can name the file whatever you want and place different resource types in one file. However, for clarity, you might want to place unique resource types in different files. For example, here are some filename conventions for resources you can create in this directory: • arrays.xml for resource arrays (typed arrays). • colors.xml for color values • dimens.xml for dimension values. • strings.xml for string values. • styles.xml for styles. See String Resources, Style Resource, and More Resource Types.
`xml/`	Arbitrary XML files that can be read at runtime by calling `Resources.getXML()` . Various XML configuration files must be saved here, such as a searchable configuration.

Accessing Resources

- You should always externalize application resources such as images and strings from your code:
  - so that you can maintain them independently
- You should also provide alternative resources for specific device configurations:
  - by grouping them in specially-named resource directories.
- At runtime, Android uses the appropriate resource based on the current configuration:
  - e.g., you might want to provide a different UI layout depending on the screen size or different strings depending on the language setting.
- Once you externalize your application resources:
  - you can access them using resource IDs that are generated in your project's R class
- Once you provide a resource in your application:
  - you can apply it by referencing its resource ID
  - all resource IDs are defined in your project's R class, which the aapt tool automatically generates


First Activity
Click me !
Second Activity
This is the second Activity

ACTIVITIES

Components Lifecycle

- Each application runs in a sandboxed environment with its own instance of an Android Runtime

- An Android application does not completely control the completion of its lifecycle:
  - hardware resources may become critically low and OS could order early termination of any process

- Components are different points through which the system can enter into an application.

- **Each component has a distinct lifecycle that defines how the component is created and destroyed.**

- **They all follow a master plan that consists of:**
  - **Begin: respond to request to instantiate them**
  - **End: when instances are destroyed**
  - **In between states: depends on the component type**

Activity Lifecycle (1/4)

- An activity can exist in essentially three states:
  - Resumed, Paused, and Stopped

- **Resumed** (or Running):
  - the activity is in the **foreground** of the screen and has user **focus**

- **Paused**:
  - another activity is in the foreground and has focus, but this one is **still visible** i.e., another activity is visible on top of the paused one, and
  - that activity is partially transparent or doesn't cover the entire screen

  - a paused activity is completely alive (the Activity object is retained in memory, it maintains all state and member information, and remains attached to the window manager), but
  - can be killed by the system in extremely low memory situations

- **Stopped**:
  - the activity is **completely obscured** by another activity (it is in the "background")
  - a stopped activity is also still alive (the Activity object is retained in memory, it maintains all state and member information, but it is not attached to the window manager)
  - however, it is no longer visible to the user, and it can be killed by the system when memory is needed elsewhere

Activity Lifecycle (2/4)

- An application usually consists of multiple activities that are loosely bound to each other

- Typically, one activity in an application is specified as the "main" activity:
  - it is presented to the user when launching the application for the first time.

- Each activity can then start another activity in order to perform different actions:
  - each time a new activity starts, the previous activity is stopped, but
  - the system preserves the activity in a stack (the "back stack")

- When a new activity starts, it is pushed onto the back stack and takes user focus

- The back stack abides to the basic "last in, first out" stack mechanism:
  - when the user is done with the current activity and presses the Back button, it is popped from the stack (and destroyed) and the previous activity resumes

Activity Lifecycle (4/4)

- To create an activity, you must:
  - create a subclass of Activity (or an existing subclass of it)
  - in your subclass, you need to implement callback methods that the system calls when the activity transitions between various states of its lifecycle

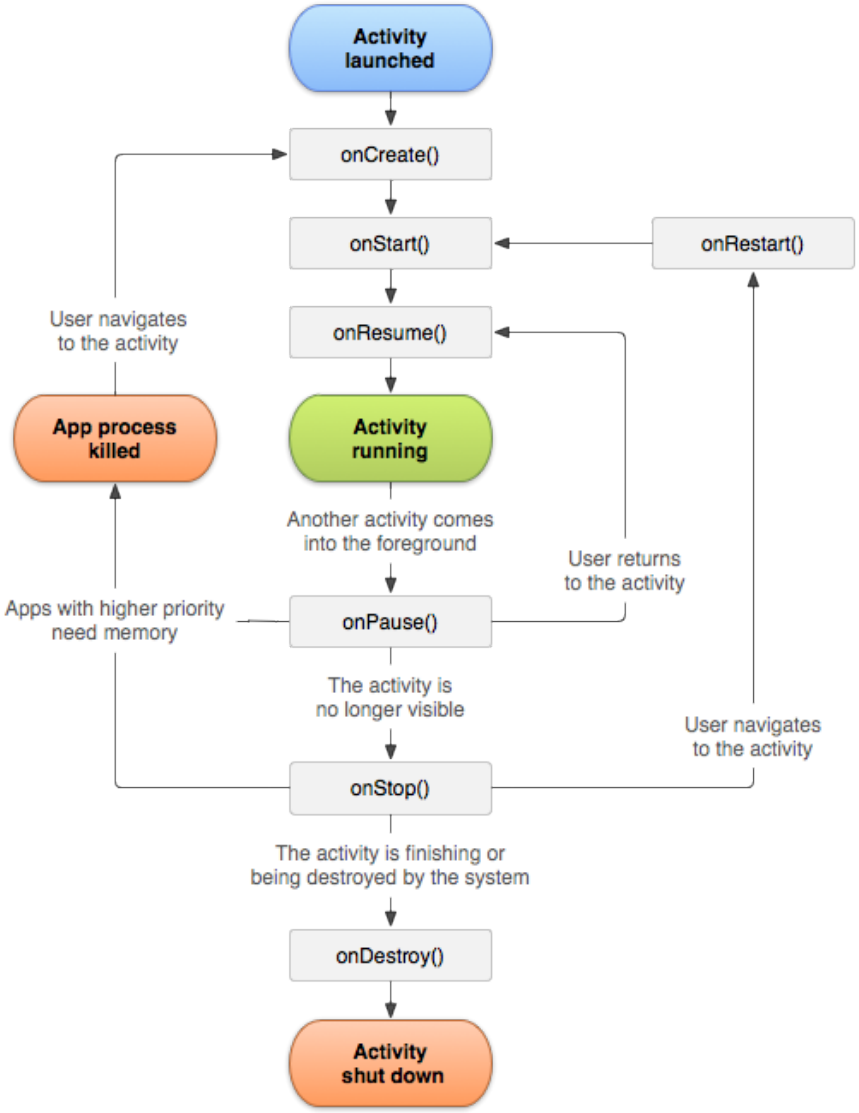
- The two most important callback methods are: onCreate and onPause

- **onCreate**:
  - you **must** implement this method
  - the system calls this when creating your activity
  - within your implementation, you should initialize the essential components of your activity
  - most importantly, this is where you must call **setContentView**() to define the layout for the activity's user interface

- **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)
  - this is usually **where you should commit** any changes that should be persisted beyond the current user session (because the user might not come back)

Activity Lifecycle Timings (1/2)

- Entire lifetime of an activity happens between:
  - the call to onCreate(), and
  - the call to onDestroy()
- Visible lifetime of an activity happens between:
  - the call to onStart(), and
  - the call to onStop()
- Foreground lifetime of an activity happens between:
  - the call to onResume(), and
  - the call to onPause()



Activity Lifecycle Methods

```
public class ExampleActivity extends Activity
{

@Override
public void onCreate(Bundle
SavedInstanceState) {
   super.onCreate(savedInstanceState);
   // The activity is being created.
}

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

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

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

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

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

Tasks and Back Stack (1/6)

- An application usually contains multiple activities

- Each activity should be designed around a specific kind of action the user can perform and can start other activities:
  - e.g., an email application might have one activity to show a list of new messages
  - when the user selects a message, a new activity opens to view that message

- An activity can even start activities that exist in other applications on the device:
  - e.g. you can define an intent to perform a "send" action and include some data, such as an email address and a message

  - an activity from another application that declares itself to handle this kind of intent then opens

  - **the intent is to send an email, so an email application's "compose" activity starts (if multiple activities support the same intent, then the system lets the user select which one to use**)

  - when the email is sent, your activity resumes and it seems as if the email activity was part of your application

  - even though the activities may be from different applications, Android maintains this seamless user experience by keeping both activities in the same *task*

Tasks and Back Stack (2/6)

- A task is a collection of activities that users interact with when performing a job

- The activities are arranged in a stack (the *back stack*), in the order in which each activity is opened

- The device Home screen is the starting place for most tasks

- When the user touches an icon in the application launcher (or a shortcut on the Home screen):
  - that application's task comes to the foreground

- If no task exists for the application (the application has not been used recently), then:
  - a new task is created and the "main" activity for that application opens as the root activity in the stack

Tasks and Back Stack (3/6)

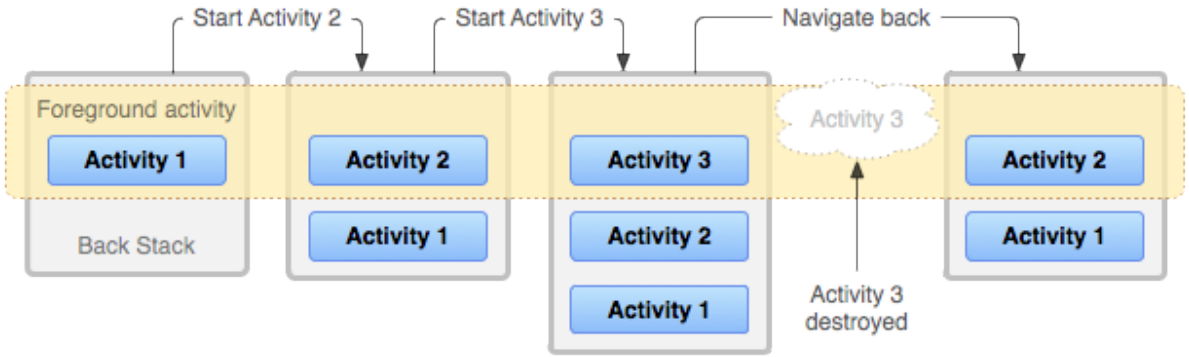
- When the current activity starts another:
  - the new activity is pushed on the top of the stack and takes focus
  - the previous activity remains in the stack, but is stopped

- When an activity stops, the system retains the current state of its user interface

- When the user presses the *Back* button:
  - the current activity is popped from the top of the stack (the activity is destroyed), and
  - the previous activity resumes (the previous state of its UI is restored)

- Activities in the stack are never rearranged, only pushed and popped from the stack:
  - pushed onto the stack when started by the current activity,
  - and popped off when the user leaves it using the *Back* button

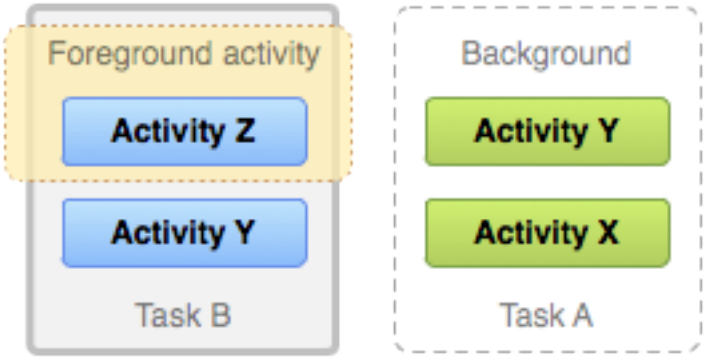
- As such, the back stack operates as a "last in, first out" object structure



Tasks and Back Stack (4/6)

- If the user continues to press *Back*, then:
  - each activity in the stack is popped off to reveal the previous one, until the user returns to the Home screen (or to whichever activity was running when the task began)

- When all activities are removed from the stack, the task no longer exists



- A task is a cohesive unit that can move to the "background" when users begin a new task or go to the Home screen, via the *Home* button

- While in the background, all the activities in the task are stopped, but the back stack for the task remains intact:
  - the task has simply lost focus while another task takes place

Tasks and Back Stack (5/6)

- A task can return to the "foreground" so users can pick up where they left off

- Suppose, for example, that the current task (Task A) has three activities in its stack:
  - the user presses the *Home* button, then starts a new application from the application launcher
  - when the Home screen appears, Task A goes into the background
  - when the new application starts, the system starts a task for that application (Task B) with its own stack of activities
  - after interacting with that application, the user returns Home again and selects the application that originally started Task A
  - now, Task A comes to the foreground—all three activities in its stack are intact and the activity at the top of the stack resumes
  - at this point, the user can also switch back to Task B by going Home and selecting the application icon that started that task (or by selecting the app's task from the overview screen)

Activities and Tasks - summary

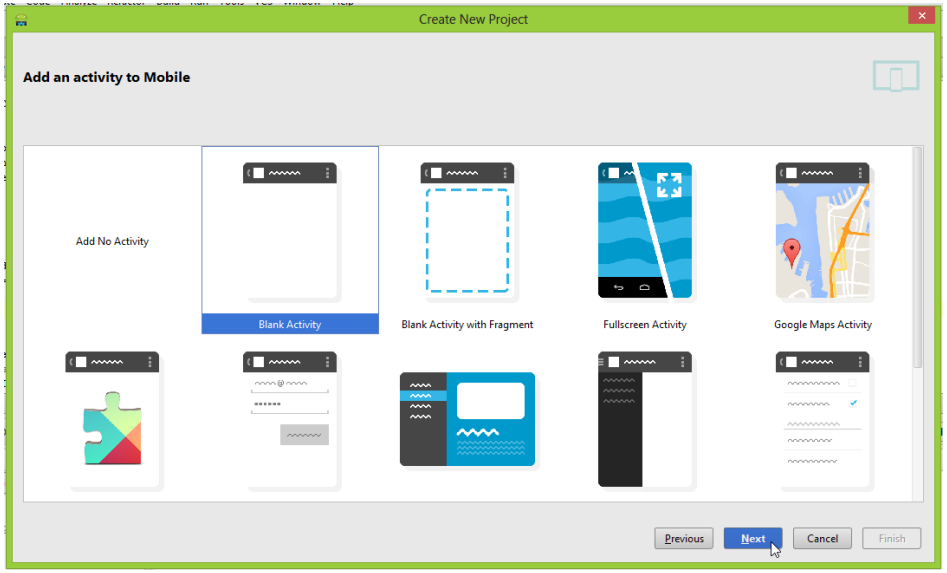
- To summarize the default behavior for activities and tasks:

- when Activity A starts Activity B, Activity A is stopped, **but the system retains its state (such as scroll position and text entered into forms)**
- if the user presses the *Back* button while in Activity B, Activity A resumes with its state restored

- when the user leaves a task by pressing the *Home* button, the current activity is stopped and its task goes into the background
- **the system retains the state of every activity in the task**
- if the user later resumes the task by selecting the launcher icon that began the task, the task comes to the foreground and resumes the activity at the top of the stack

- if the user presses the *Back* button, the current activity is popped from the stack and destroyed
- the previous activity in the stack is resumed
- when an activity is destroyed, the system *does not* retain the activity's state

- Activities can be instantiated multiple times, even from other tasks.



CREATING ACTIVITIES

Creating and Activity and its Interface (1/2)

- To create an activity:
  - create a subclass of `Activity` (or an existing subclass of it)
  - in the subclass, implement callback methods that the system calls when the activity transitions between various states of its lifecycle (e.g. when the activity is being created, stopped, resumed, or destroyed)

- The user interface for an activity is provided by a hierarchy of views:
  - these are objects derived from the `View` class
  - each view controls a particular rectangular space within the activity's window (it responds the user)
  - e.g. a view might be a button that initiates an action when the user touches it

- Provides a number of ready-made views that can be used to design and organize the layout:
  - "Layouts" are views derived from `ViewGroup`
  - they provide a unique layout model for its child views (e.g. linear layout, grid layout)
  - you can also subclass the `View` and `ViewGroup` classes (or existing subclasses) to create your own widgets and layouts and apply them to your activity layout

Creating and Activity and its Interface (2/2)

- The most common way to define a layout using views is with an XML layout file saved in your application resource:

  - this way, you can maintain the design of your user interface separately from the source code that defines the activity's behavior

  - you can set the layout as the UI for your activity with `setContentView()`, passing the resource ID for the layout

  - you can also create new `Views` in your activity code and build a view hierarchy by inserting new `Views` into a `ViewGroup`, then use that layout by passing the root `ViewGroup` to `setContentView()`

- "Widgets" are views that provide a visual (and interactive) elements for the screen, such as a button, text field, checkbox, or just an image

Declaring an Activity

- Declare your activity in the manifest file for it to be accessible to the system:
  - open your manifest file and add an `<activity>` element as a child of the `<application>` element
  - e.g. :

```
<manifest ... >
  <application ... >
     <activity android:name=".ExampleActivity" />
     ...
  </application ... >
  ...
</manifest>
```

- There are several other attributes that you can include in this element to define properties:
  - e.g. the label for the activity, an icon for the activity, or a theme to style the activity's UI

- The `android:name` attribute is the only required attribute:
  - it specifies the class name of the activity

- Once you publish your application, you should not change this name:
  - if you do, you might break some functionality, such as application shortcuts

Intent Filters (1/2)

- An `<activity>` element can also specify various intent filters in order to declare how other application components may activate it:
  - simply use the `<intent-filter>` element

- When you create a new application (using the Android SDK tools):
  - the stub activity that's created automatically includes an intent filter
  - this filter declares that the activity responds to the "main" action and should be placed in the "launcher" category

```
<activity android:name=".ExampleActivity"
android:icon="@drawable/app_icon">
   <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category
android:name="android.intent.category.LAUNCHER" />
   </intent-filter>
</activity>
```

- The `<action>` element specifies that this is the "main" entry point to the application

- The `<category>` element specifies that this activity should be listed in the system's application launcher (to allow users to launch this activity)

Intent Filters (2/2)

- If you intend for your application to be self-contained and not allow other applications to activate its activities:
  - you don't need any other intent filters
  - only one activity should have the "main" action and "launcher" category, as in the previous example
  - activities that you don't want to make available to other applications should have no intent filters and you can start them yourself using explicit intents

- If you want your activity to respond to implicit intents that are delivered from other applications (and your own):
  - you must define additional intent filters for your activity
  - for each type of intent to which you want to respond, you must include an
    `<intent-filter>` that includes an
    `<action>` element and, optionally, a
    `<category>` element and/or a
    `<data>` element
  - these elements specify the type of intent to which your activity can respond

Starting an Activity (1/2)

- You can start an activity by:
  - calling `startActivity()`, and
  - passing it an Intent that describes the activity you want to start
  - the intent specifies:
    - either the exact activity you want to start, or
    - describes the type of action you want to perform (and the system selects the appropriate activity for you, which can even be from a different application)
  - an intent can also carry small amounts of data to be used by the activity that is started

- When working within your own application, you'll often need to simply launch a known activity:
  - you can do so by creating an intent that explicitly defines the activity you want to start, using the class name.
  - e.g. one activity starts another activity named `SignInActivity`

```
Intent intent = new Intent(this,
SignInActivity.class);
startActivity(intent);
```

Starting an Activity (2/2)

- If your application wants to perform some action using data from your activity, and it does not have its own activities to perform such actions:
  - e.g. send an email, text message, or status update
  - you can instead leverage the activities provided by other applications on the device, which can perform the actions for you
  - you can create an intent that describes an action you want to perform and the system launches the appropriate activity from another application
  - e.g., if you want to allow the user to send an email message, you can create the following intent:

```
Intent intent = new Intent(Intent.ACTION_SEND);
intent.putExtra(Intent.EXTRA_EMAIL,
recipientArray);
startActivity(intent);
```

  - The `EXTRA_EMAIL` added to the intent:
    - string array of email addresses to which the email should be sent
    - when an email application responds to this intent, it reads the string array provided in the extra and places them in the "to" field of the email composition form
    - in this situation, the email application's activity starts and when is done, your activity resumes

Starting an Activity for a Result (1/3)

- You can also start another activity and receive a result back

- To receive a result:
  - call `startActivityForResult()` (instead of `startActivity()`)
  - e.g. your app can start a camera app and receive the captured photo as a result

- The activity that responds must be designed to return a result:
  - when it does, it sends the result as another Intent object
  - your activity receives it in the `onActivityResult()` callback

Starting an Activity for a Result (2/3)

- There's nothing special about the `Intent` object you use when starting an activity for a result:
  - you do need to pass an additional integer argument to the `startActivityForResult()` method

- The integer argument is a "request code“:
  - it identifies your request
  - when you receive the result Intent, the callback provides the same request code so that your app can properly identify the result and determine how to handle it

- E.g., start an activity that allows the user to pick a contact:

```
static final int PICK_CONTACT_REQUEST = 1;  // The request code
...
private void pickContact() {
  Intent pickContactIntent = new Intent(Intent.ACTION_PICK,
Uri.parse("content://contacts"));
  // Show only contacts w/ phone numbers
  pickContactIntent.setType(Phone.CONTENT_TYPE);
  startActivityForResult(pickContactIntent,
PICK_CONTACT_REQUEST);
}
```

Starting an Activity for a Result (3/3)

- When the user is done with the subsequent activity and returns:
  - the system calls your activity's on`ActivityResult()` method
  - this method includes three arguments:
    - the request code you passed to `startActivityForResult()`
    - a result code specified by the second activity:
      - this is either `RESULT_OK` if the operation was successful, or
      - `RESULT_CANCELED` if the user backed out or the operation failed for some reason.
    - an Intent that carries the result data

- E.g., handle the result for the "pick a contact" intent:
  - the result `Intent` returned by Android's Contacts or People app provides a content `Uri` that identifies the contact the user selected

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
   // Check which request we're responding to
   if (requestCode == PICK_CONTACT_REQUEST) {
      // Make sure the request was successful
      if (resultCode == RESULT_OK) {
         // The user picked a contact.
         // The Intent's data Uri identifies which contact was selected.
         // Do something with the contact here
      }
   }
}
```

Shutting Down an Activity

- You can shut down an activity by calling its `finish()` method.
- You can also shut down a separate activity that you previously started by calling `finishActivity()`.
- The Android system manages the life of an activity for you, so you do not need to finish your own activities.
- Calling these methods could adversely affect the expected user experience:
  - thus, it should only be used when you absolutely do not want the user to return to this instance of the activity

SERVICES

Services

- A service is a component that:
  - runs in the background
  - to perform long-running operations
  - without needing to interact with the user

- A service can essentially take two states:

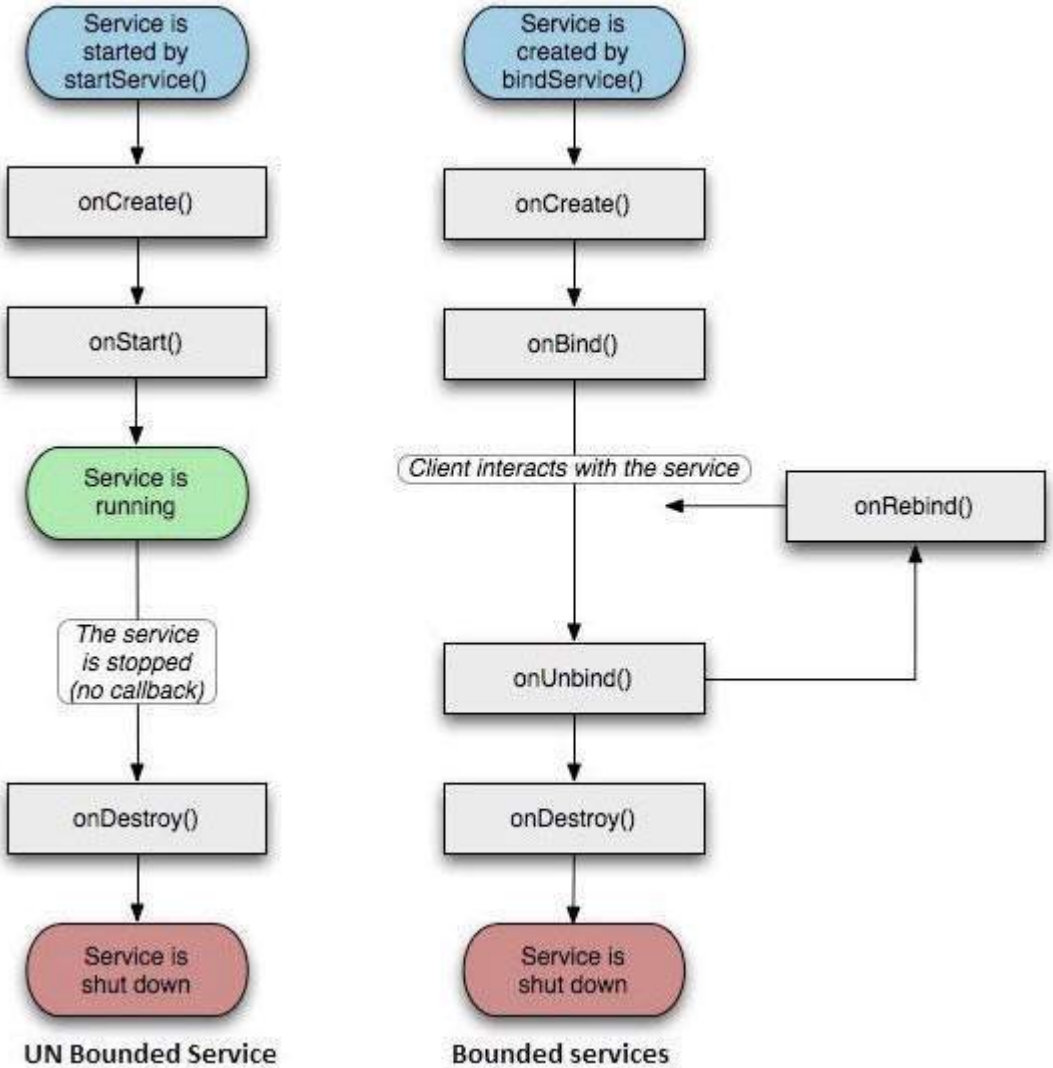
Started	A service is **started** when an application component, such as an activity, starts it by calling *startService()*. Once started, a service can run in the background indefinitely, even if the component that started it is destroyed.
Bound	A service is **bound** when an application component binds 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).

Services Lifecycle

- A service has life cycle callback methods that you can implement to:
  - monitor changes in the service's state, and

- Left: life cycle when the service is created with `startService()`

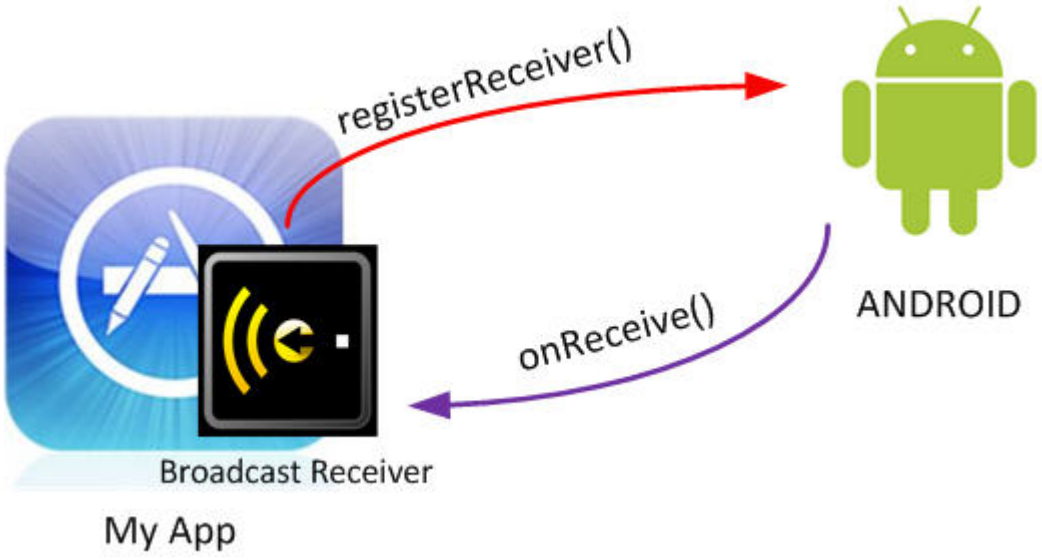
- Right: life cycle when the service is created with `bindService()`



Creating a Service

- 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
- You don't need to implement all the callbacks methods:
  - it's important that you understand each one and implement those that ensure your app behaves the way users expect

Callback	Description
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.
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*.
onUnbind()	The system calls this method when all clients have disconnected from a particular interface published by the service.
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)*.
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.
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.



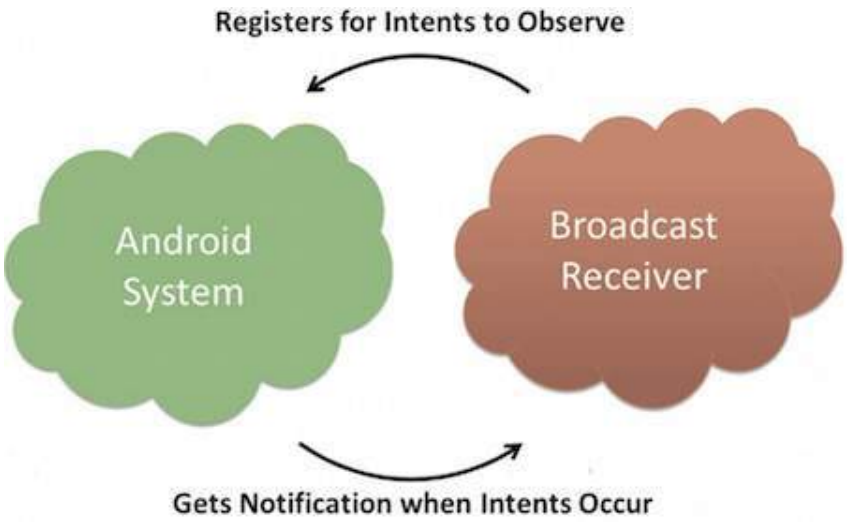
BROADCAST RECEIVERS

Broadcast Receivers

- Broadcast Receivers simply respond to broadcast messages from other applications or from the system itself.

- For example:
  - Applications can also initiate broadcasts to let other applications know that some data has been downloaded to the device exists, and is available for them to use,
  - so that the broadcast receiver intercepts this communication and initiates an appropriate action

- There are two important steps to make BroadcastReceiver works for the system broadcast intents:
  - Creating the Broadcast Receiver
  - Registering Broadcast Receiver



Creating and Registering a Broadcast Receiver

- A broadcast receiver is implemented:
  - subclass of BroadcastReceiver class, and
  - overriding the onReceive() method where each message is received as an 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();
   }
}
```

A toast is a view containing a quick little message for the user.

- An application listens for specific broadcast intents by:
  - registering a broadcast receiver in AndroidManifest.xml file

- E.g. registering MyReceiver for system generated event ACTION_BOOT_COMPLETED:
  - fired by the system once the Android system has booted

```
<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>
```

System Events

Event Constant	Description
android.intent.action.BATTERY_CHANGED	Sticky broadcast containing the charging state, level, and other information about the battery.
android.intent.action.BATTERY_LOW	Indicates low battery condition on the device.
android.intent.action.BATTERY_OKAY	Indicates the battery is now okay after being low.
android.intent.action.BOOT_COMPLETED	This is broadcast once, after the system has finished booting.
android.intent.action.BUG_REPORT	Show activity for reporting a bug.
android.intent.action.CALL	Perform a call to someone specified by the data.
android.intent.action.CALL_BUTTON	The user pressed the "call" button to go to the dialer or other appropriate UI for placing a call.
android.intent.action.DATE_CHANGED	The date has changed.
android.intent.action.REBOOT	Have the device reboot.

Broadcasting Custom Intents

- An application can generate and send custom intents:
  - Create and send those intents by using the sendBroadcast() method.

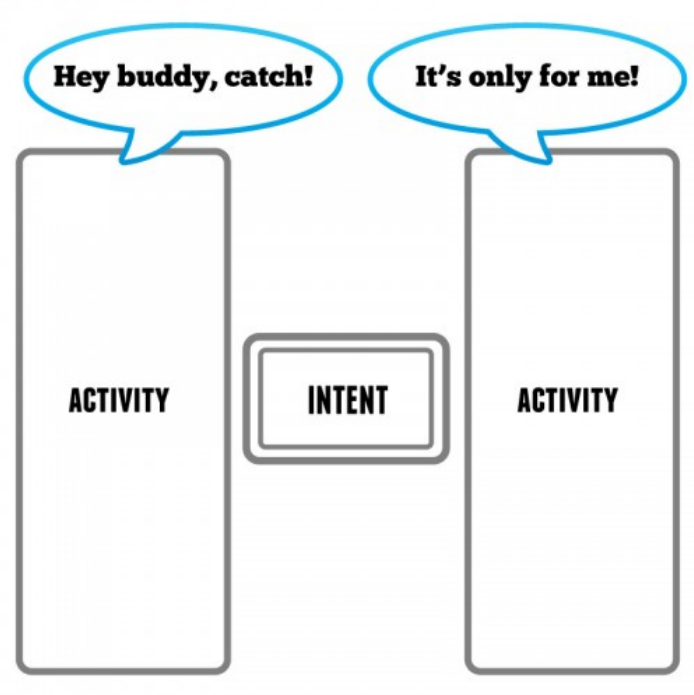
```
public void broadcastIntent(View view)
{
   Intent intent = new Intent();
   intent.setAction("com.tutorialspoint.CUSTOM_INTENT");
   sendBroadcast(intent);
}
```

- The intent `com.tutorialspoint.CUSTOM_INTENT` can also be registered as a broadcast message (like system-wide intents).

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

      <intent-filter>
         <action android:name="com.tutorialspoint.CUSTOM_INTENT">
         </action>
      </intent-filter>

   </receiver>
</application>
```



INTENTS

Intents and Intent Filters (1/2)

- An Intent is a messaging object you can use to request an action from another app component

- An intent can be seen as abstract description of an operation to be performed; thus it can be used with:
  - startActivity to launch an Activity,
  - startService(Intent) or bindService(Intent, ServiceConnection, int) to communicate with a background Service
  - broadcastIntent to send it to any interested BroadcastReceiver components, and

> The intent itself, an Intent object, is a passive data structure holding an abstract description of an operation to be performed

- Start an activity:
  - an Activity represents a single screen in an app
  - start a new instance of an Activity by passing an Intent to startActivity()
  - the Intent describes the activity to start and carries any necessary data
  - if you want to receive a result from the activity when it finishes, call startActivityForResult()
  - your activity receives the result as a separate Intent object in your activity's onActivityResult() callback

Intents and Intent Filters (2/2)

- Start a service:
  - a Service is a component that performs operations in the background without a user interface
  - you can start a service to perform a one-time operation (such as download a file) by passing an Intent to startService()
  - the Intent describes the service to start and carries any necessary data
  - if the service is designed with a client-server interface, you can bind to the service from another component by passing an Intent to bindService()
- A broadcast is a message that any app can receive:
  - the system delivers various broadcasts for system events (e.g. when the system boots up or the device starts charging)
  - e.g. you can deliver a broadcast to other apps by passing an Intent to sendBroadcast()

Intent Types (1/2)

- There are two types of intents:
  - explicit and implicit

- Explicit intents:
  - Specify the component to start by name (the fully-qualified class name)
  - You'll typically use an explicit intent to start a component in your own app, because you know the class name of the activity or service you want to start
  - E.g., start a new activity in response to a user action or start a service to download a file in the background.

- Implicit intents:
  - do not name a specific component, but instead declare a general action to perform, which allows a component from another app to handle it
  - e.g., if you want to show the user a location on a map, you can use an implicit intent to request that another capable app show a specified location on a map

Intent Types (2/2)

- When you create an explicit intent to start an activity or service:
  - the system immediately starts the app component specified in the Intent object

- When you create an implicit intent:
  - the system finds the appropriate component to start by comparing the contents of the intent to the intent filters declared in the manifest file of other apps on the device
  - if the intent matches an intent filter, the system starts that component and delivers it the Intent object
  - if multiple intent filters are compatible, the system displays a dialog so the user can pick which app to use

Example: email Intent

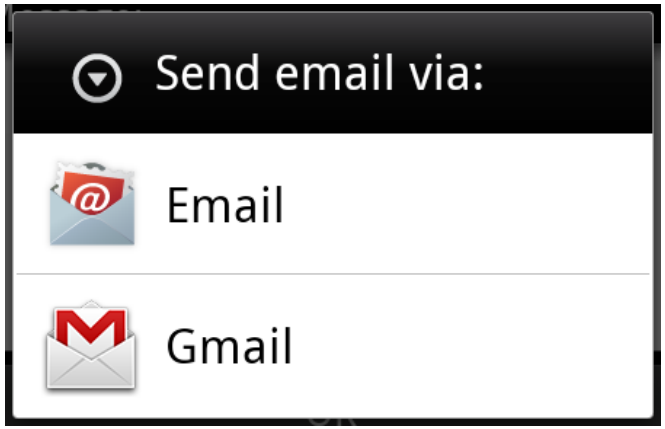
- You have an Activity that needs to launch an email client and sends an email using your Android device

- For this purpose, your Activity would send an ACTION_SEND along with appropriate chooser, to the Android Intent Resolver

- The specified chooser gives the proper interface for the user to pick how to send your email data

```
Intent email = new Intent(Intent.ACTION_SEND, Uri.parse("mailto:"));
email.putExtra(Intent.EXTRA_EMAIL, recipients);
email.putExtra(Intent.EXTRA_SUBJECT, subject.getText().toString());
email.putExtra(Intent.EXTRA_TEXT, body.getText().toString());
startActivity(Intent.createChooser(email, "Choose an email client from..."));
```

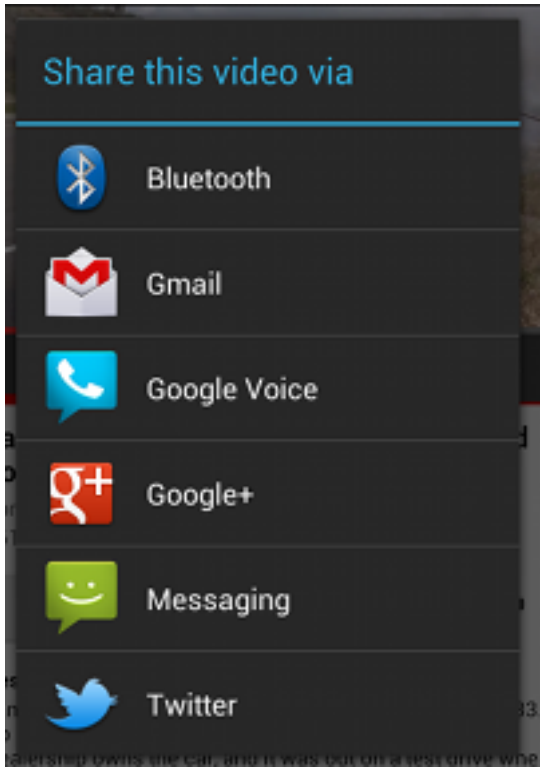
- It calls the startActivity method to start an email activity and result should be as shown:



Forcing an App Chooser

- When there is more than one app that responds to your implicit intent:
  - the user can select which app to use, and
  - make that app the default choice for the action

- If multiple apps can respond to the intent, the user might want to use a different app each time:
  - you should explicitly show a chooser dialog
  - the chooser dialog asks the user to select which app to use for the action every time



Explicit Intent

- An explicit intent is one that:
  - you use to launch a specific app component, such as a particular activity or service in your app
  - to create an explicit intent, define the component name for the Intent object—all other intent properties are optional

- Example of a service (in your app) named DownloadService:
  - designed to download a file from the web,
  - you can start it with the following code:

```
// Executed in an Activity, so 'this' is the Context
// The fileUrl is a string URL, such as "http://www.example.com/image.png"
Intent downloadIntent = new Intent(this, DownloadService.class);
downloadIntent.setData(Uri.parse(fileUrl));
startService(downloadIntent);
```

- The Intent(Context, Class) constructor supplies the app Context and the component a Class object:
  - thus, this intent explicitly starts the DownloadService class in the app

Implicit Intent

- An implicit intent specifies an action that can invoke any app on the device able to perform the action

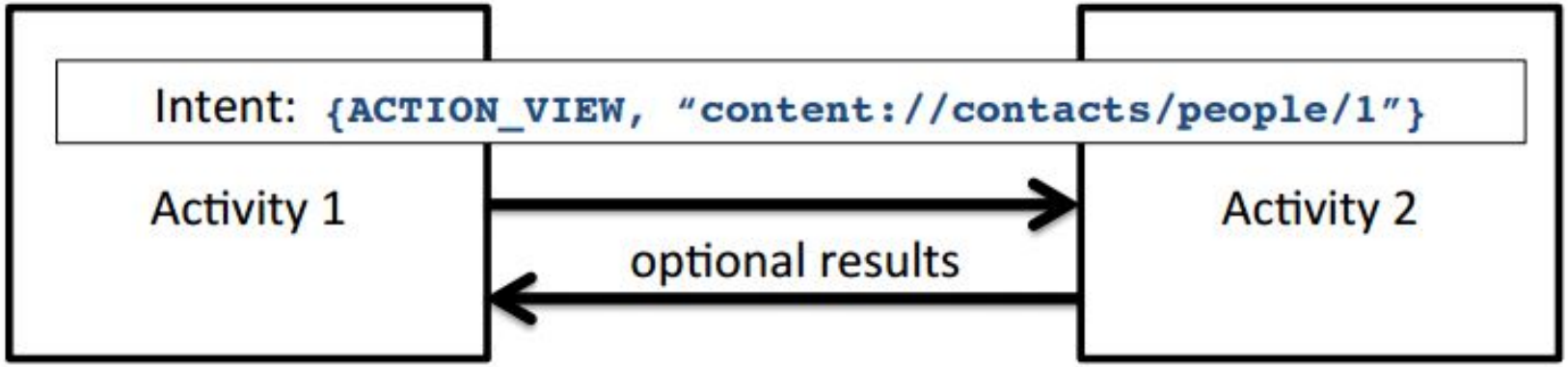
- Using an implicit intent is useful when:
  - your app cannot perform the action, but
  - other apps probably can and you'd like the user to pick which app to use

- E.g., if you have content you want the user to share with other people:
  - create an intent with the ACTION_SEND action, and
  - add extras that specify the content to share
  - when you call startActivity() with that intent, the user can pick an app through which to share the content

Main Arguments of Intents

- Action: the built-in action to be performed, such as ACTION_VIEW, or user-created-action

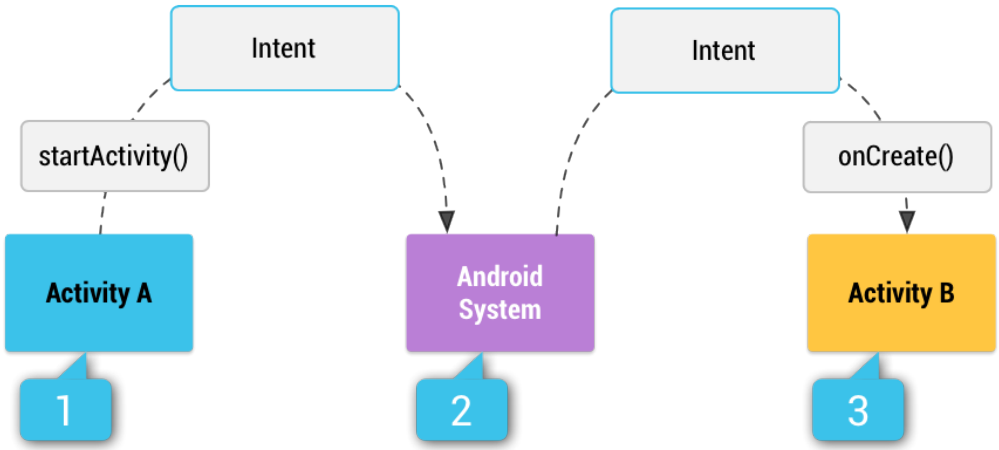
- Data: the primary data to operate on, such as a phone number to be calles (expressed as a URI)



Intent Filter

- An intent filter is an expression in an app's manifest file that specifies the type of intents that the component would like to receive:
  - e.g., by declaring an intent filter for an activity, you make it possible for other apps to directly start your activity with a certain kind of intent
  - likewise, if you do *not* declare any intent filters for an activity, then it can be started only with an explicit intent

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



Building an Intent

- An `Intent` object carries information that the Android system uses to determine which component to start:
  - the exact component name or component category that should receive the intent, plus
  - information that the recipient component uses in order to properly perform the action (such as the action to take and the data to act upon)

- The primary information contained in an `Intent` is the following:
  - Component name
  - Action
  - Data
  - Category
  - Extras
  - Flags

Intent – component

- Component name:
  - the name of the component to start
  - this is optional, but it's the critical piece of information that makes an intent **explicit**,
  - it means that the intent should be delivered only to the app component defined by the component name
  - without a component name, the intent is **implicit** (the system decides which component should receive the intent based on the other intent information, e.g. action, data, and category)

- Action:
  - a string that specifies the generic action to perform
  - in the case of a broadcast intent, this is the action that took place and is being reported
  - the action largely determines how the rest of the intent is structure (particularly what is contained in the data and extras)
  - you can specify your own actions for use by intents within your app (or for use by other apps to invoke components in your app), but
  - you should usually use action constants defined by the Intent class or other framework classes
  - Example:
    - `ACTION_VIEW` - use this action in an intent with `startActivity()` when you have some information that an activity can show to the user (e.g. a photo to view in a gallery app, or an address to view in a map app)
    - `ACTION_SEND` - also known as the "share" intent, you should use this in an intent with `startActivity()` when you have some data that the user can share through another app, such as an email app or social sharing app

Intent – data and category

- Data:
  - the URI (a Uri object) that references the data to be acted on and/or the MIME type of that data
  - the type of data supplied is generally dictated by the intent's action
  - e.g., if the action is ACTION_EDIT, the data should contain the URI of the document to edit
  - when creating an intent, it's often important to specify the type of data (its MIME type) in addition to its URI
  - e.g., an activity that's able to display images probably won't be able to play an audio file, even though the URI formats could be similar
  - so specifying the MIME type of your data helps the Android system find the best component to receive your intent
  - however, the MIME type can sometimes be inferred from the URI—particularly when the data is a content: URI, which indicates the data is located on the device and controlled by a ContentProvider, which makes the data MIME type visible to the system

- Category:
  - a string containing additional information about the kind of component that should handle the intent
  - any number of category descriptions can be placed in an intent, but most intents do not require a category
  - some common categories:
    - CATEGORY_BROWSABLE - the target activity allows itself to be started by a web browser to display data referenced by a link—such as an image or an e-mail message
    - CATEGORY_LAUNCHER - the activity is the initial activity of a task and is listed in the system's application launcher