# Introduction to Android Security

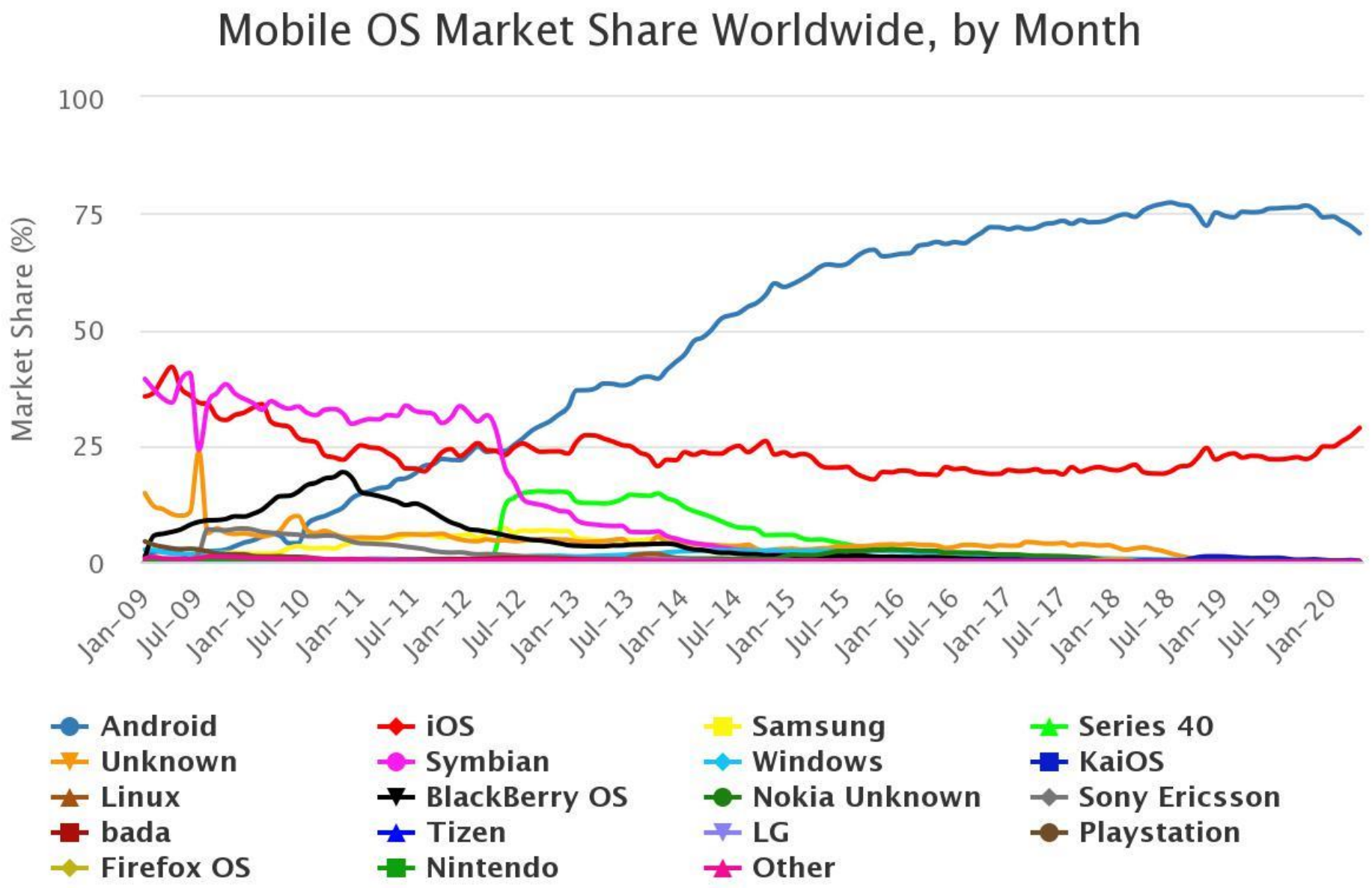
| | | |
|---|---|---|
| Introduction to Android | Android Architecture, Android Run-time | Android Application Framework |
| Android Application Components, Sandboxing | Application Inter-Process Communication | App Permission, Android Boot Process, Partition, File Systems |

# Mobile OS






# Mobile OS Market Share World-Wide




https://trends.google.com/trends/explore?date=2010-03-03%202022-04-03&q=%2Fm%2F02wxtgw,%2Fm%2F03wbl14,%2Fm%2F0bbv_rf,%2Fm%2F0jgf9,Windows%20Mobile

# Introduction to Android

- Android is an **open-source operating system** based on **Linux** with a **Java and kotlin** programming interface for mobile devices such as Smartphone (Touch Screen Devices who supports Android OS) as well for Tablets too.
- Android was developed by the Open Handset Alliance **(OHA)**, which is led by **Google**.
- The Open Handset Alliance (OHA) is a consortium of multiple companies like Samsung, Sony, Intel and many more to provide services and deploy handsets using the android platform.

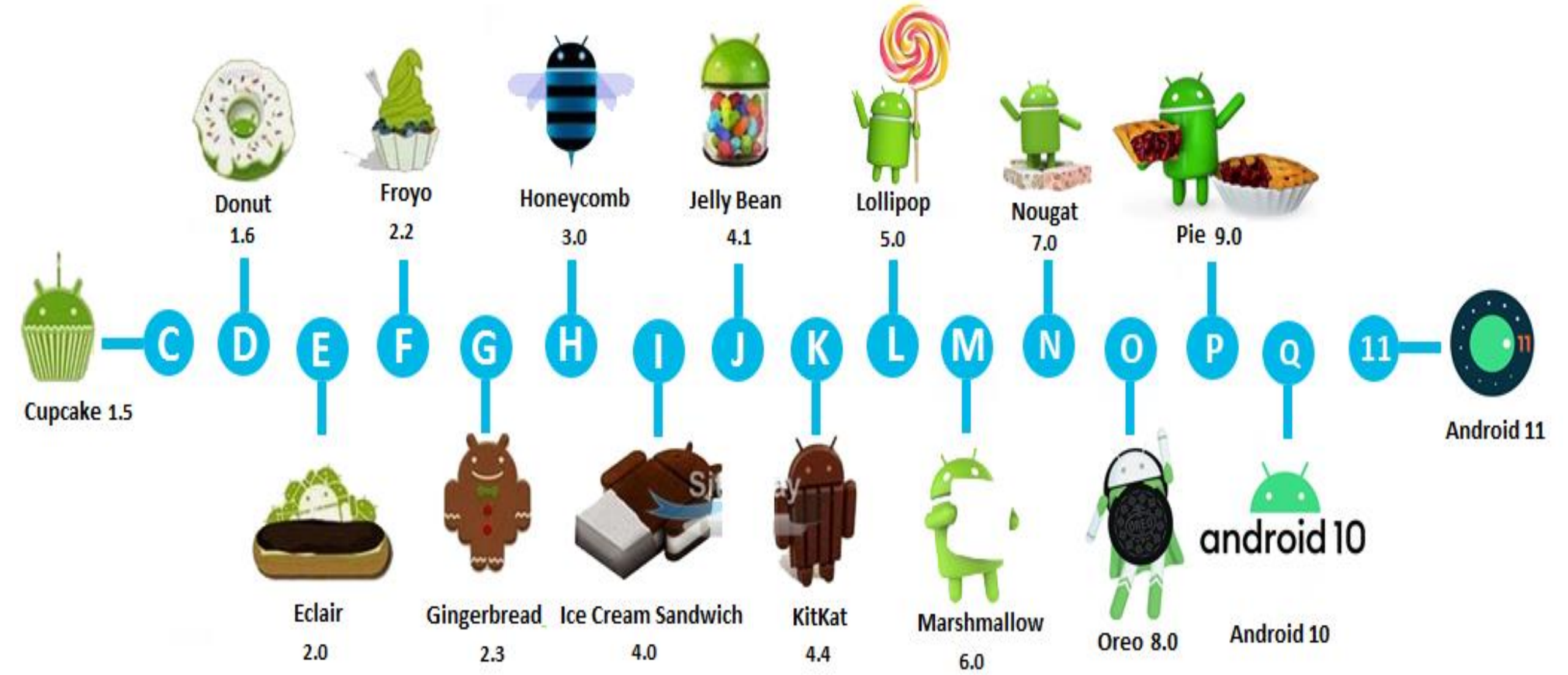
# Introduction to Android

- In 2007, Google released a first beta version of the Android Software Development Kit (SDK) and the first commercial version of Android 1.0 (with name Alpha), was released in September 2008.
- In 2012, Google released another version of android, 4.1 Jelly Bean.
- In 2014, Google announced another Version, 5.0 Lollipop.
- Latest release 12, Released in Oct, 2021 and on the way to release Android 13 by June or July

# Introduction to Android

- Initially, Andy Rubin founded Android Incorporation in Palo Alto, California, United States in October, 2003.
- In 17th August 2005, Google acquired android Incorporation. Since then, it is in the subsidiary of Google Incorporation.
- The key employees of Android Incorporation are Andy Rubin, Rich Miner, Chris White and Nick Sears.
- Originally intended for camera but shifted to smart phones later because of low market for camera only.
- Android is the nick name of Andy Rubin given by coworkers because of his love to robots.
- In 2007, Google announces the development of android OS. 7) In 2008, HTC launched the first android mobile.

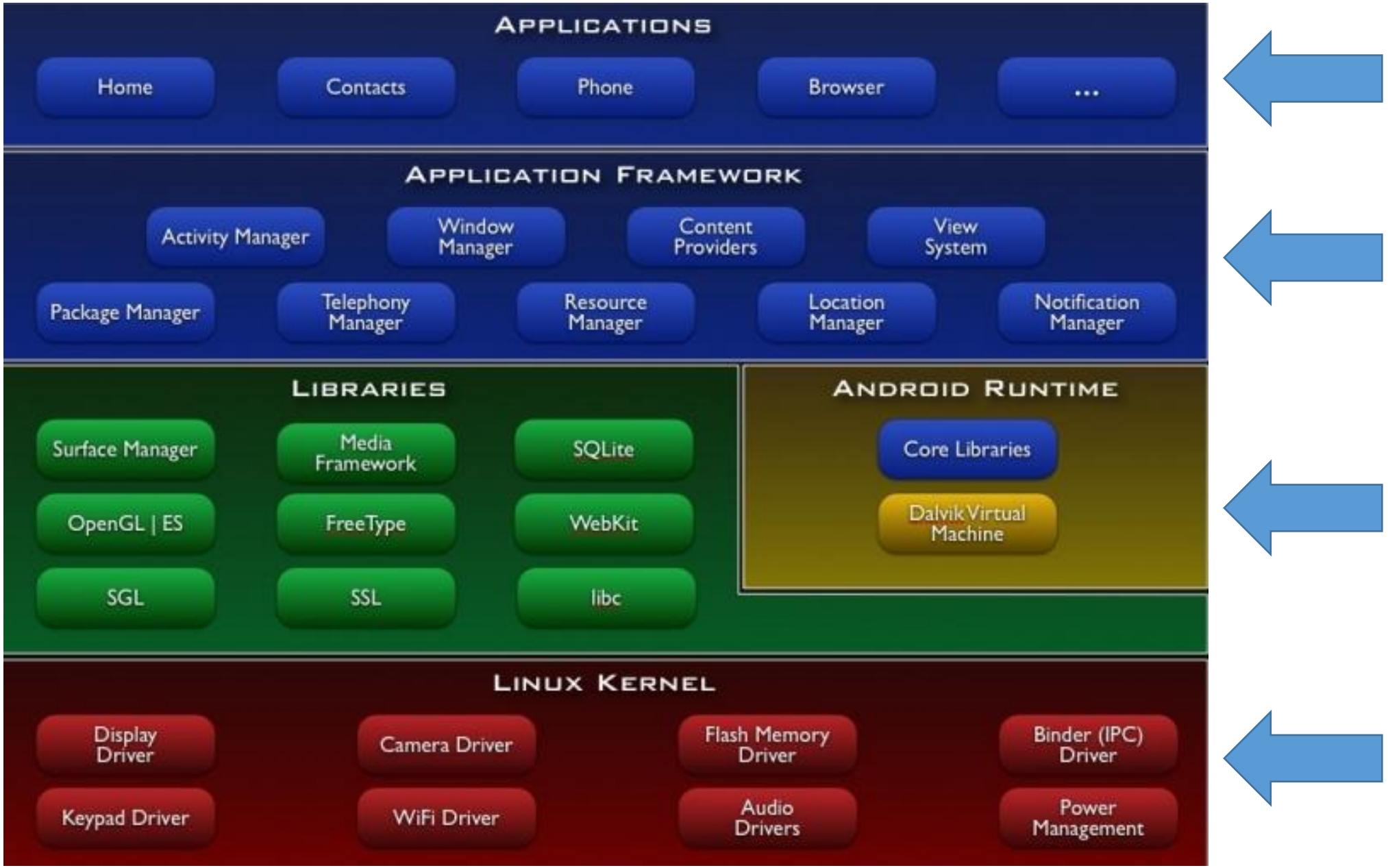
# Android Versions



# Android Architecture

- Any operating system (desktop or mobile) takes responsibility for **managing the resources** of the system and provides a way for **applications to talk to hardware or physical components** in order to accomplish certain tasks.
- OS manage mobile phones, manages **memory** and **processes, enforces security**, takes care of networking issues
- The Android operating system consists of a stack of layers running on top of each other.

# Android Architecture



# Android Architecture

## ✓The Linux kernel:

- Provides a level of **abstraction** between the device hardware and the upper layers.
- Kernel contains **drivers to understand the hardware instruction.**
- The drivers in the kernel control the underlying hardware.
- As shown in the preceding figure, the kernel contains drivers related to Wi-Fi, Bluetooth, USB, audio, display, and so on.
- Such as **process management, memory management, security, and networking, are managed** by Linux kernel

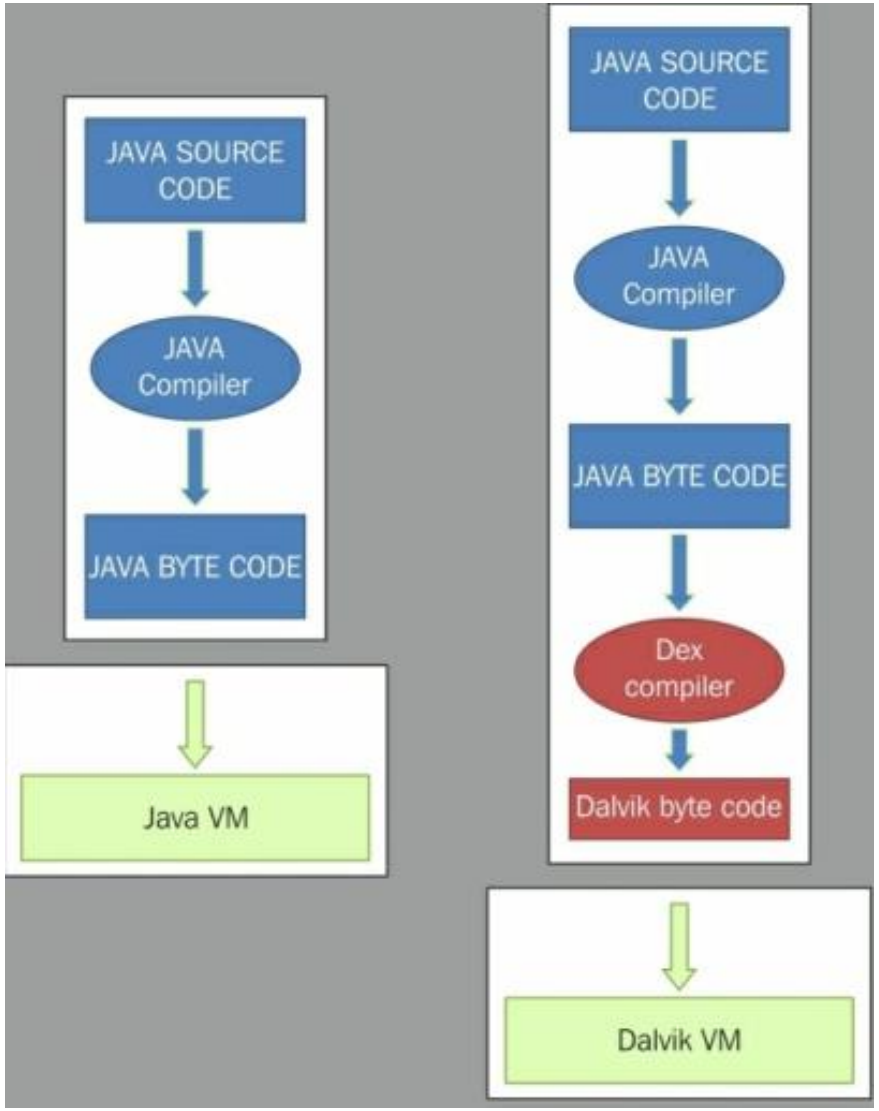
# Android Architecture

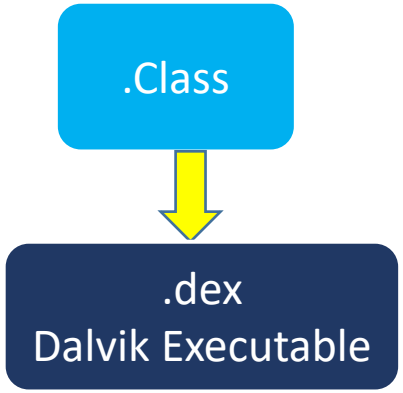
**Libraries:**

- On top of Linux kernel are Android's native libraries.
- It is with the help of these **libraries that the device handles different types of data**.
- These libraries are written in the **C or C++ programming languages** and are specific to a particular hardware.
- For example, the media framework library supports the recording and playback of audio, video and picture formats.

# Android Architecture

- Dalvik Virtual Machine



# Android Architecture

## Dalvik Virtual Machine

**JVM Bytecode**

```
public static int add(int, int);
  Code:
    0: iload_0❶
    1: iload_1❷
    2: iadd❸
    3: ireturn❹
```

**Dalvik Bytecode**

```
.method public static add(II)I

    add-int v0, p0, p1❺


    return v0❻
.end method
```

- Here, the JVM uses two instructions to load the parameters onto the stack **(1** and **2)**, then executes the addition **3**, and finally returns the result **4**.
- In contrast, Dalvik uses a single instruction to add parameters (in registers p0 and p1) and puts the result in the v0 register **5**. Finally, it returns the contents of the v0 register **6**.
- As you can see, Dalvik uses fewer instructions to achieve the same result.
- Generally speaking, register-based VMs use fewer instructions, but the resulting code is larger than the corresponding code in a stack-based VM.

# Android Architecture

✓**Dalvik byte code :**

- Dalvik byte code is an **optimized byte code** suitable for low-memory and low-processing environments.
- Also, note that JVM's byte code consists of **one or more .class files**, depending on the number of Java files that are present in an application, but Dalvik byte code is composed of **only one .dex file.**

# Android Application Framework

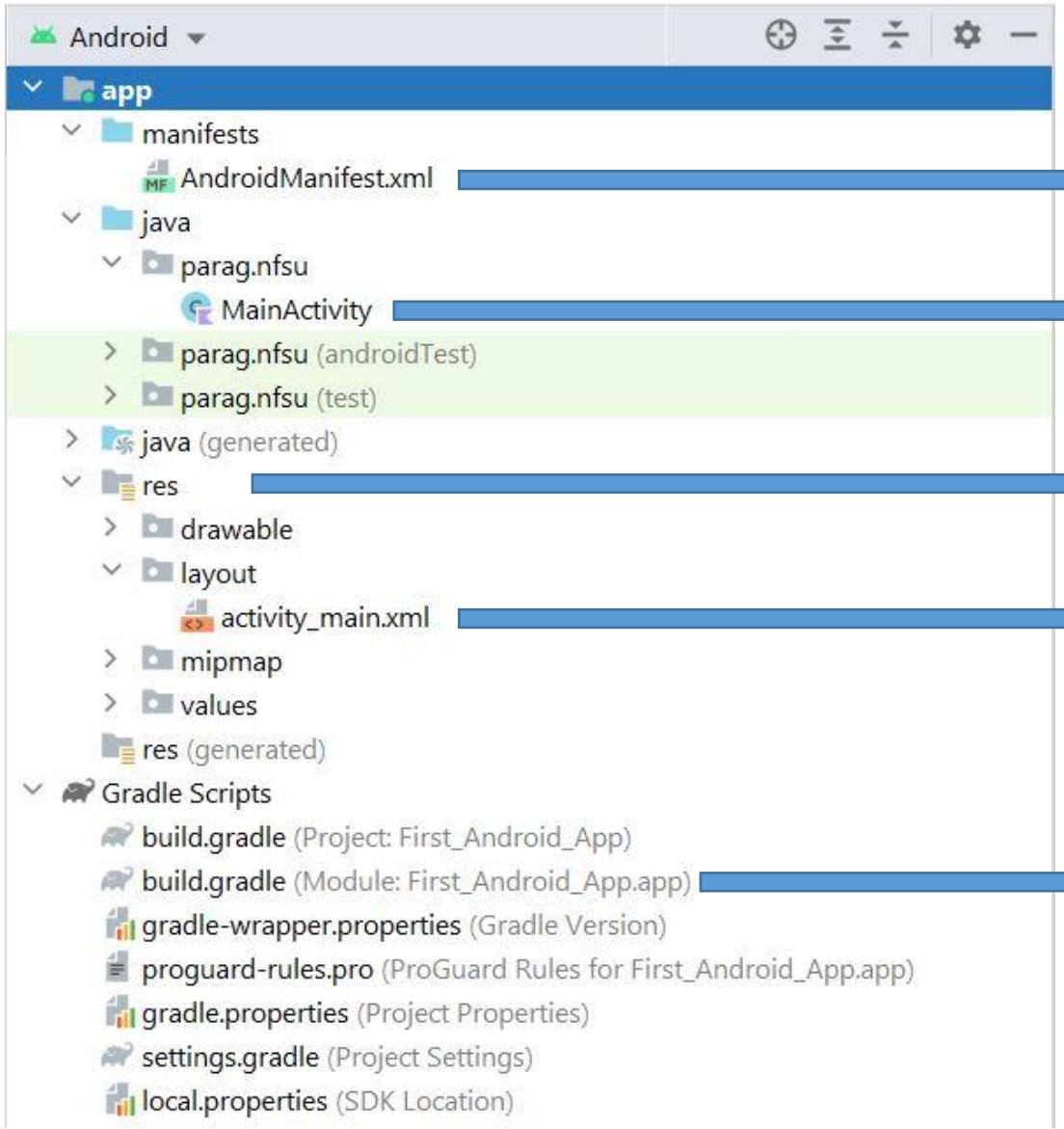
## Framework Managers

| FRAMEWORK SERVICE | DESCRIPTION |
|---|---|
| Activity Manager | Manages Intent resolution/destinations, app/activity launch, and so on |
| View System | Manages views (UI compositions that a user sees) in activities |
| Package Manager | Manages information and tasks about packages currently and previously queued to be installed on the system |
| Telephony Manager | Manages information and tasks related to telephony services, radio state(s), and network and subscriber information |
| Resource Manager | Provides access to non-code app resources such as graphics, UI layouts, string data, and so on |
| Location Manager | Provides an interface for setting and retrieving (GPS, cell, WiFi) location information, such as location fix/coordinates |
| Notification Manager | Manages various event notifications, such as playing sounds, vibrating, flashing LEDs, and displaying icons in the status bar |

# Android Application Framework

- Framework Managers
- Android Application Folder Structure
- AndroidManifest.xml
- Resources
- Android Application Components
  - Activity
  - Service
  - Broadcast Receiver
  - Content Providers

# Android Application Framework




**Android Application Folder Structure**

1

2

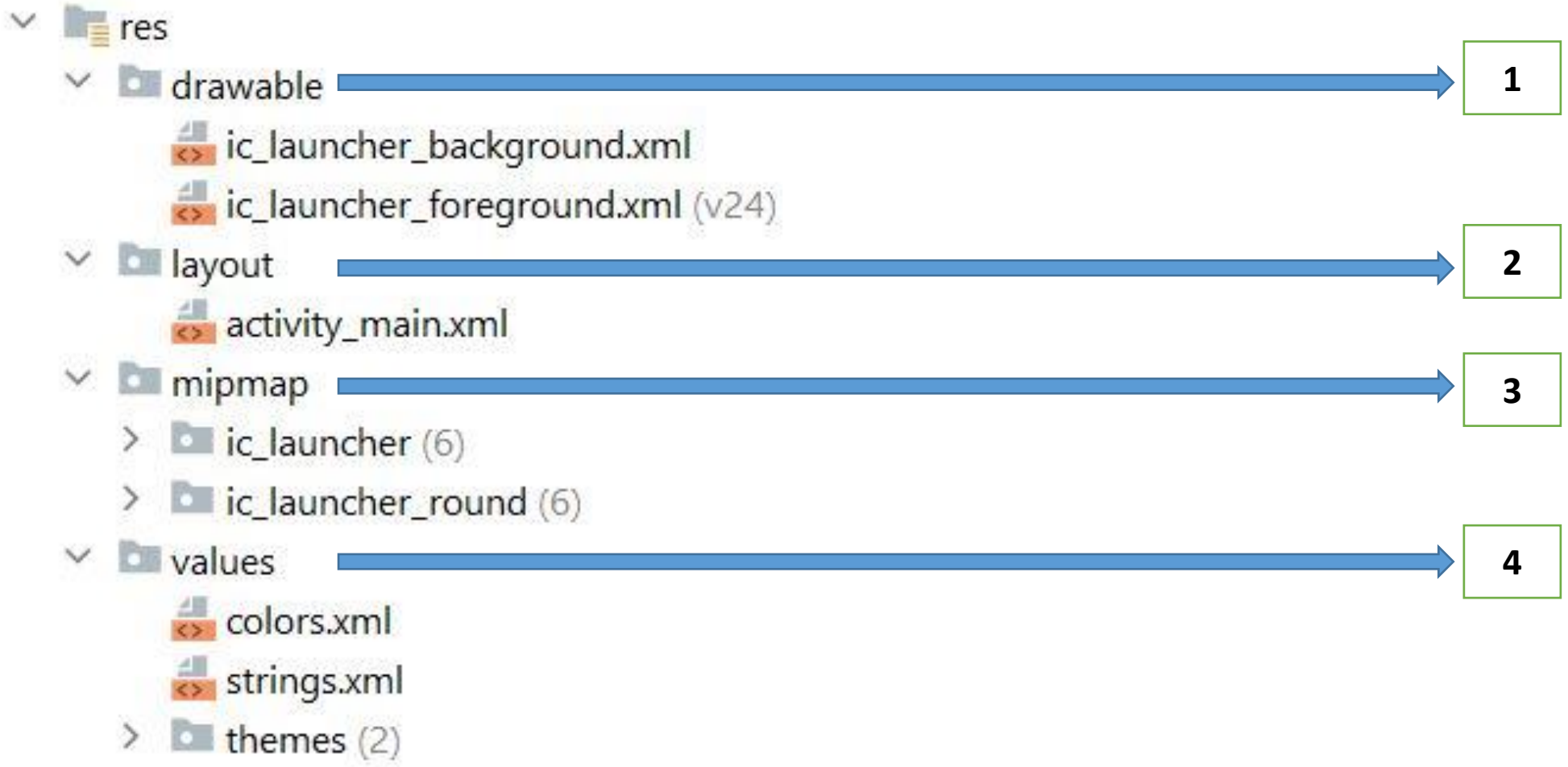
3

4

5

# Android Application Framework

## Application Resources Overview



# Android Application Framework

**AndroidManifest.xml**

- Every app project must have an AndroidManifest.xml file (with precisely that name) at the **root of the project** source set.
- The manifest file describes essential information about your app to the **Android build tools, the Android operating system, and Google Play.**
- **Manifest file is required to declare the following:**
  - The components of the app, which include all **activities, services, broadcast receivers, and content providers.**

# Android Application Framework

## AndroidManifest.xml

- **Manifest file is required to declare the following:**
  - The app's package name, which usually matches your **code's namespace**. The **Android build** tools use this to determine the location of code entities when building your project.
  - The **permissions** that the app needs in order to access protected parts of the system or other apps.
  - The manifest file is also where you can declare what **types of hardware or software features your app requires**, and thus, which types of devices your app is compatible with.

# Android Application Framework

AndroidManifest.xml

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

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

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

</manifest>
```

**AndroidManifest.xml**

**1 - Which other tag we can use here?**

**<activity>elements for activities**
**<service> elements for services**
**<receiver> elements for broadcast receivers**
**<provider> elements for content providers**

# Android Application Framework

**Android Application Components**

- Four main component that can be used within an Android Application
  - **Activity –** UI and handle user integration with mobile phone.
  - **Service –** Background processing associated with android application.
  - **Broadcast Receivers –** Handle communication between Android OS and Application.
  - **Content Providers** - handle data and database managements operations.

# Android Application Framework
# Android Components

**Activities**

- Your application's presentation layer.
- The UI of your application is built around one or more extensions of the Activity class.
- Activities use Fragments and Views to layout and display information, and to respond to user actions. Compared to desktop development, Activities are equivalent to Forms.

**Services**

- The invisible workers of your application.
- Service components run without a UI, updating your data sources and Activities, triggering Notifications, and broadcasting Intents.
- They're used to perform long running tasks, or those that require no user interaction (such as download file, music player, alarm etc.)

**Content Providers**

- Shareable persistent data storage.
- Content Providers manage and persist application data and typically interact with SQL databases.
- They're also the preferred means to share data across application boundaries.
- You can configure your application's Content Providers to allow access from other applications, and you can access the Content Providers exposed by others.
- Android devices include several native Content Providers that expose useful databases such as the media files, contact, call logs, calendar etc

# Android Application Framework
# Android Components

**Intents**

- A powerful inter application message passing framework.
- Intents are used extensively throughout Android.
- You can use Intents to start and stop Activities and Services, to broadcast messages system-wide or to an explicit Activity, Service, or Broadcast Receiver, or to request an action be performed on a particular piece of data.
- **Two types**
- Implicit Intent
- Explicit Intent

**Broadcast Receiver**

- Intent listeners
- Broadcast Receivers enable your application to listen for Intents that match the criteria you specify.
- Broadcast Receivers start your application to react to any received Intent, making them perfect for creating event-driven applications.

# Android Application Framework

## Application Resources Overview

- Resources are the **additional files and static content** that your code uses, such as **bitmaps, layout definitions, user interface strings,** animation instructions, and more.
- drawable/Bitmap files (.png, .jpg, .gif etc)
- layout/XML files that define a user interface layout.
- menu/XML files that define app menus, such as an Options Menu, Context Menu, or Sub Menu.
- values/XML files that contain simple values, such as strings, integers, and colors.

# Android Application Framework

## Activity

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="First Android App"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/Theme.FirstAndroidApp">
    <activity
        android:name=".MainActivity"
        android:exported="true">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</application>
```

- An activity represents a single screen with a user interface just like window, frame or form.
- An activity represents a **single screen** with a **user interface** just like window or frame.
- An application can have **one or more activities** without any restrictions.
- Every activity you **define** for your application must be declared in your **AndroidManifest.xml file** and
- the main activity for your app must be declared in the manifest with an **<intent-filter>** that includes the **MAIN action and LAUNCHER** category.

# Android Application Framework
# Activity

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/ap
    package="parag.nfsu">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="First Android App"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.FirstAndroidApp">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

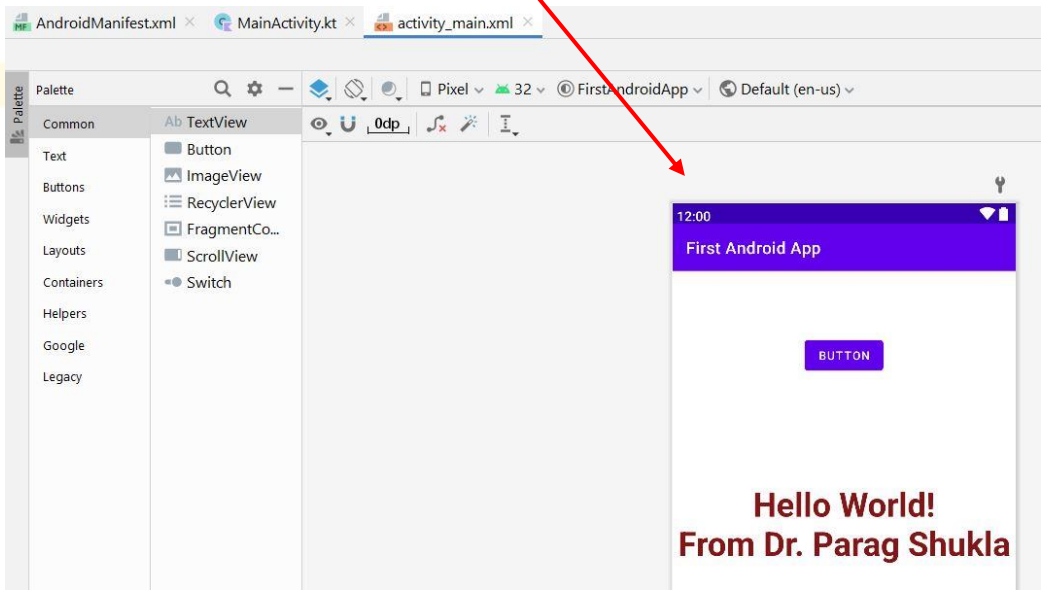
AndroidManifest.xml | MainActivity.kt | activity_main.xml

```kotlin
package parag.nfsu

import ...

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        var t1 = findViewById<TextView>(R.id.textView)
        var b1 = findViewById<Button>(R.id.button)
        b1.setOnClickListener { it: View!
            t1.setText("NFSU - Gandhinagar")
        }
    }
}
```



# Android Application Framework
## Intent

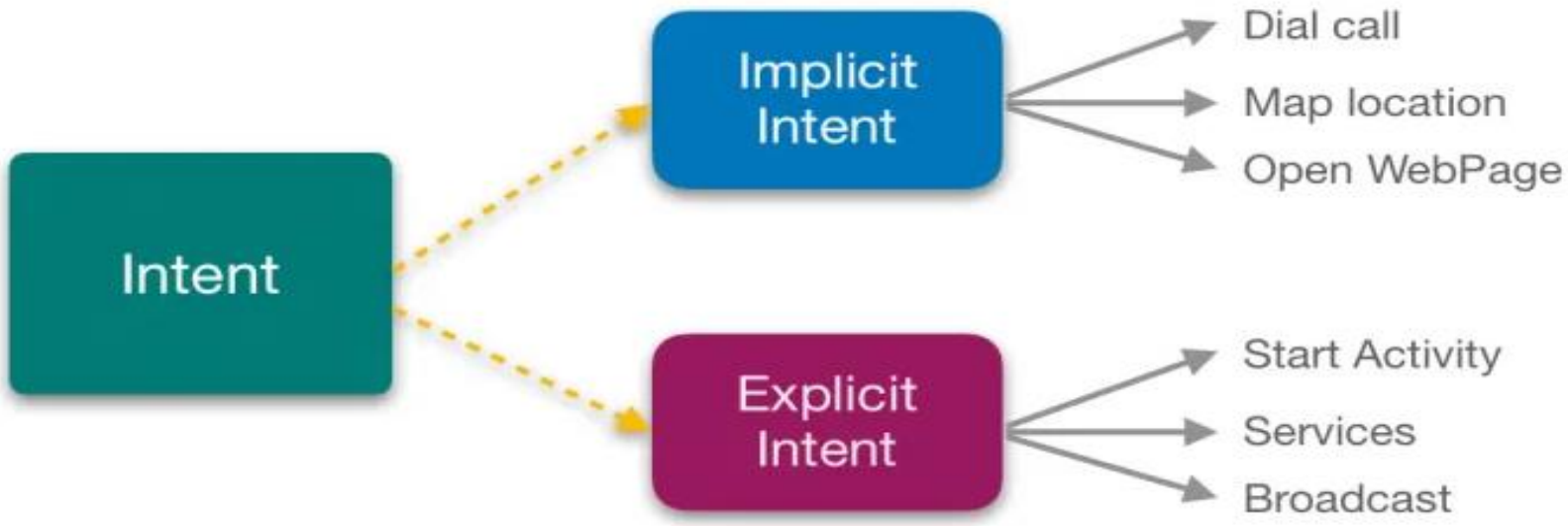
- An Intent is a **messaging object** you can use to request an action from another app component.
- Although intents facilitate **communication** between components in several ways, there are three fundamental use cases:
- It can be used with
  - **startActivity to launch an Activity,**
  - **broadcastIntent** to send it to any interested **BroadcastReceiver** components, and
  - **startService(Intent)** or **bindService(Intent, ServiceConnection, int)** to communicate with a background Service.

# Android Application Framework
## Intent

- An Intent object can contain the following components
  1. **Action:** This is mandatory part of the Intent object and is a string naming the action to be performed.
  2. **Category :** The category is an optional part of Intent object and it's a string containing additional information about the kind of component that should handle the intent.

# Android Application Framework
# Intent

- There are two types of intents:
  1. **Explicit** This intent satisfies the request within the application component. It takes the fully qualified class name of activities or services that we want to start.
  2. **Implicit Intent:** This intent does not specify the component name. It invokes the component of another app to handle it.



# Android Application Framework
# Intent

- There are two types of intents:

1. **Explicit** This intent satisfies the request within the application component. It takes the fully qualified class name of activities or services that we want to start.

   Syntax : Starting the Activity

   **Java**

   ```
   Intent intent = new Intent(getApplicationContext(), ActivityTwo.class);
   startActivity(intent);
   ```

   **Kotlin**

   ```
   var intent = Intent(applicationContext, MainActivity::class.java)
   startActivity(intent)
   ```

# Android Application Framework
# Intent

Syntax:

**Starting the Service**

```
Intent intent = new Intent(this, HelloService.class);
startService(intent);
```

**Delivering Broadcast Receive**

```
Intent intent = new Intent (“unique name”);
context.sendBroadcast(intent);
```

**2. Implicit Intent:** This intent does not specify the component name. It invokes the component of another native app to handle it.

Syntax

**Starting the Activity**

```
Intent intent = new Intent();
intent.setAction=(Intent.ACTION_DIAL);
intent.setAction=(Intent.ACTION_VIEW);
intent.setAction=(Intent.ACTION_CALL);
```

# Android Application Framework
# Intent

```
fun openMap(view: View) {
    var ed = findViewById<EditText>(R.id.edLocation)
    var uri = Uri.parse("geo:0,0?q="+ed.text.toString())
    startActivity(Intent(Intent.ACTION_VIEW,uri))
}
fun shareText(view: View) {
    var ed = findViewById<EditText>(R.id.edText)
    ShareCompat.IntentBuilder.from(this)
        .setText(ed.text.toString())
        .setChooserTitle("Share Text")
        .setType("text/plain")
        .startChooser()
}
fun openWebsite(view: View) {
    var ed = findViewById<EditText>(R.id.edWebsite)
    var uri = Uri.parse(ed.text.toString())
    startActivity(Intent(Intent.ACTION_VIEW,uri))
}
fun openDialPad(view: View) {
    var ed = findViewById<EditText>(R.id.edPhone)
    var uri = Uri.parse("tel:"+ed.text.toString())
    startActivity(Intent(Intent.ACTION_DIAL,uri))
}
```

**To Open Google Map**

**To Share Text**

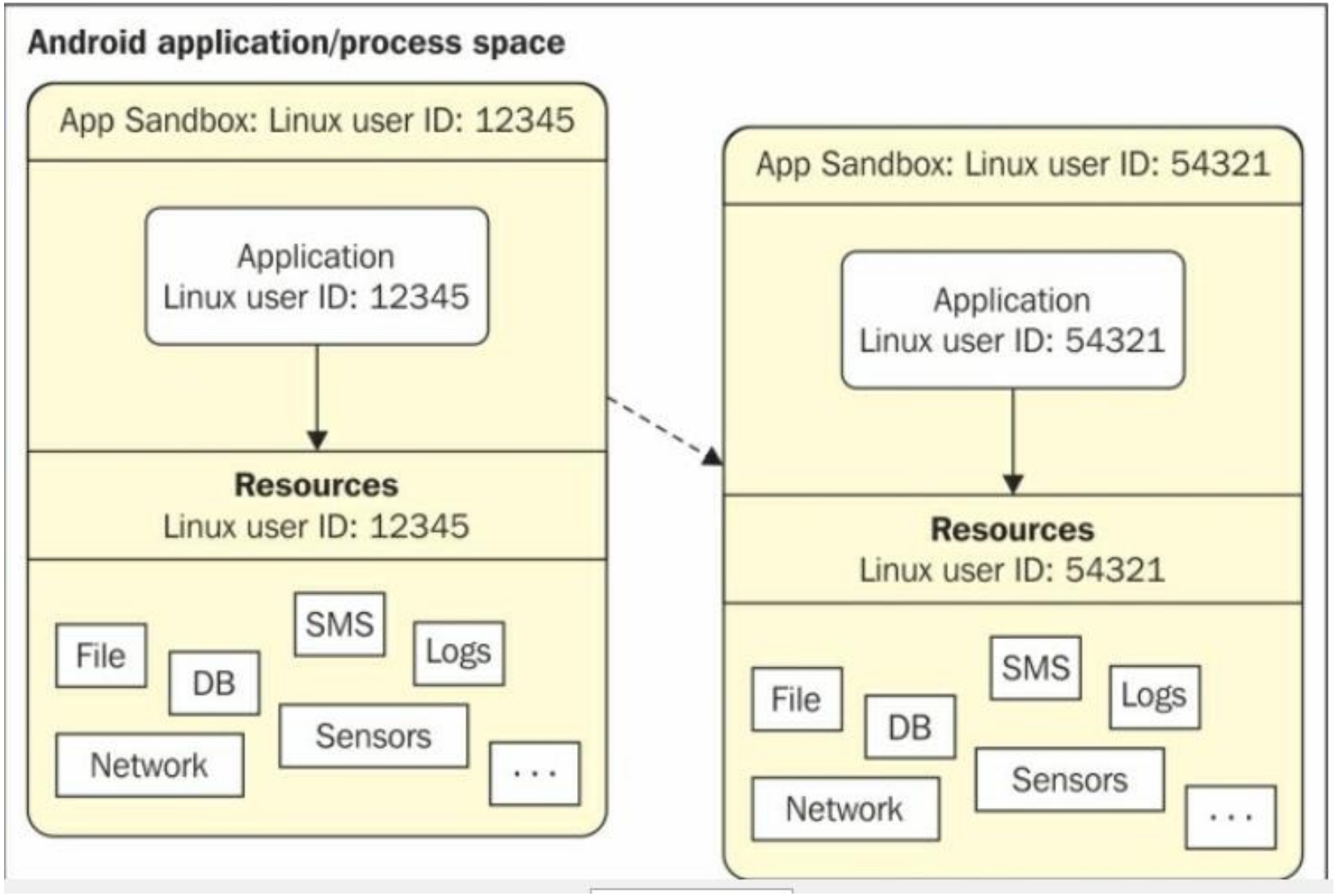
**To Open Browser**

**To Open Dial Pad**

# Application Sandboxing

- In order to isolate applications from each other
- Android takes advantage of the **Linux user-based** protection model.
- In Linux systems, each user is assigned a **unique user ID (UID)** and users are segregated so that one user does not access the data of another user.
- All resources under a particular user are run with the same privileges.
- Android application is assigned a **UID and is run** as a separate process.
- What this means is that even if an installed application tries to do something **malicious**, it can do it only within its **context and with the permissions it has.**
- By default, applications cannot **read or access the data** of other applications and have limited access to the operating system.

# Application Sandboxing




*Two applications on different processes on with different UID's*

# Application Sandboxing

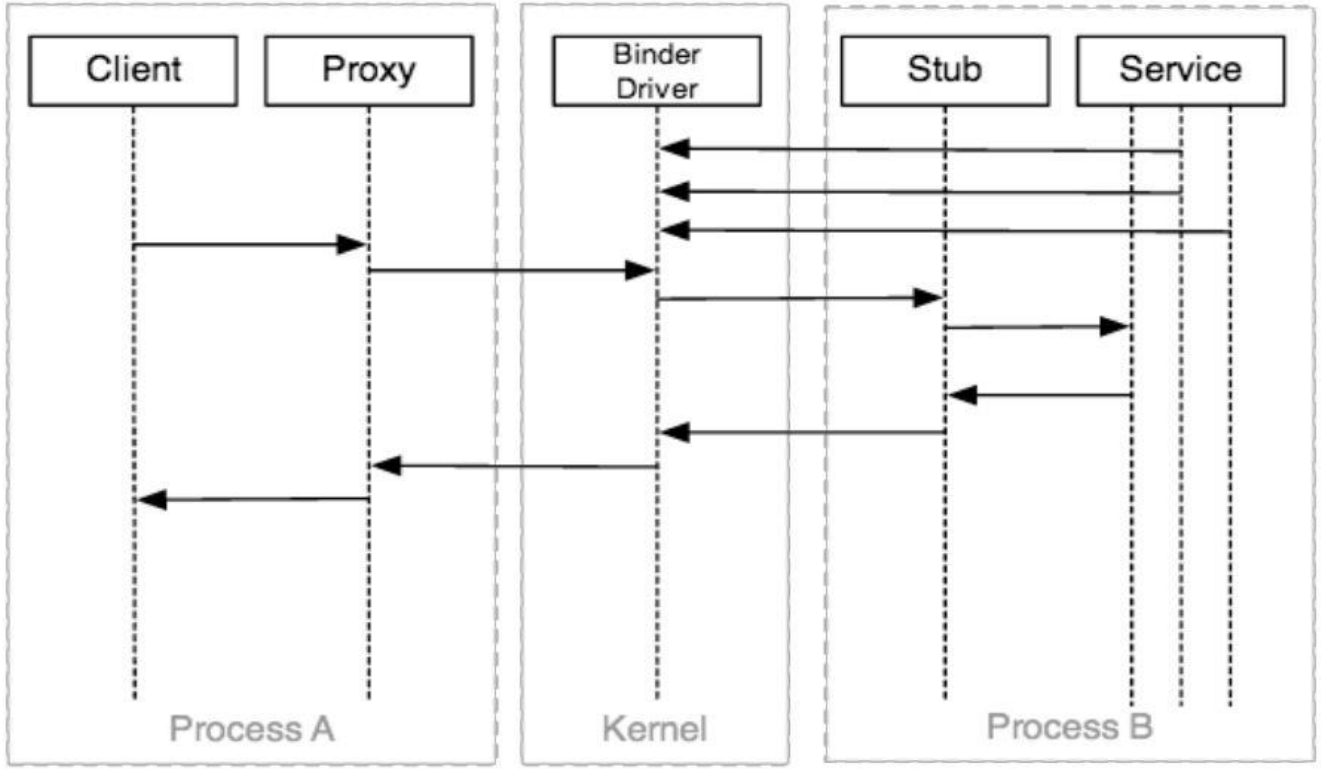
- Since the application sandbox mechanism is implemented at the **kernel level,** it applies to both **native applications and OS applications.**

# Secure Inter Process Communication

- As discussed, sandboxing of the apps is achieved by running apps in different processes with different Linux identities.
- System services run in separate processes and have more privileges.
- Thus, in order to organize data and signals between these processes, an inter-process communication (IPC) framework is needed.
- In Android, this is achieved with the use of the Binder
- The **Binder framework** in Android provides the capabilities required to organize **all types of communication between various processes**.
- Android application components, such as **intents** and **content providers**, are also built on top of this Binder framework.
- Using this framework, it is possible to perform a variety of actions such as invoking methods on remote objects.

# Secure Inter Process Communication

- Let us suppose the application in Process 'A' wants to use certain behavior exposed by a service which runs in Process 'B'.
- In this case, Process 'A' is the client and Process 'B' is the service.
- The communication model using Binder is shown in the following diagram:




**Binder Communication Model**

# Secure Inter Process Communication

- All communication between the processes using the Binder framework occurs through the **/dev/binder Linux kernel driver**.
- The permissions to this **device driver are set to word readable** and writable.
- Hence, any application may write to and read from this device driver.
- All communications between the client and server happen through proxies on the client side and **stubs** on the **server side**.

# Secure Inter Process Communication

- To get this process working, **each service** must be registered with the context manager.
- the **context manager** acts as a **name service**, providing the handle of a service using the name of this service.
- Thus, a client needs to know only the **name of a service** to communicate.
- Each service (also called a Binder service) exposed using the **Binder mechanism** is assigned with a token.

# Secure Inter Process Communication

- This token is a 32-bit value and is unique across all processes in the system.
- A client can start interacting with the service after discovering this value.
- This is possible with the help of Binder’s context manager.
- The context manager acts as a name service, providing the handle of a service using the name of this service
- The name is resolved by the context manager and the client receives the token that is later used for communicating with the service

# Application Permission

- The purpose of a **permission is to protect the privacy of an Android user.**
- Android apps must **request permission to access sensitive user data** (such as **contacts and SMS**), as well as certain system features (such as **camera and internet**).
- Depending on the feature, the system might grant the permission **automatically** or might **prompt** the user to approve the request.
- A central design point of the Android security architecture is that **no app, by default, has permission to perform any operations** that would adversely impact other apps, the operating system, or the user.
- This includes **reading or writing the user's private data** (such as contacts or emails), reading or writing another app's files, performing network access, keeping the device awake, and so on.

# Application Permission

- An app must publicize the permissions it requires by including **<uses-permission>** tags in the app manifest.
- For example, an app that needs to **send SMS messages** would have this line in the manifest:

```
<manifest        xmlns:android=http://schemas.android.com/apk/res/android
 package="parag.nfsu.telephonyApp">
<uses-permission android:name="android.permission.SEND_SMS"/>
<application...>
        ...
    </application>
</manifest>
```

# Application Permission Approval

- If your app lists **normal permissions** in its manifest (that is, permissions that **don't pose much risk** to the user's privacy or the device's operation), the system automatically grants those permissions to your app.
- If your app lists **dangerous permissions** in its manifest (that is, permissions that could potentially affect the user's privacy or the device's normal operation), such as the **SEND_SMS permission above**, the user must explicitly agree to grant those permissions.
- **Protection Level**
  - Permissions are divided into several **protection levels.**
  - The protection level affects whether runtime permission requests are required.
  - There are three protection levels that affect **third-party apps: normal, signature, and dangerous permissions.**

# Application Permission Protection Level

1. **Normal permissions:**

- Normal permissions cover areas where **your app needs to access data or resources outside the app's sandbox**, but where there's very **little risk** to the user's privacy or the operation of other apps.
- For example, permission to **set the time zone** is a normal permission.
- If an app declares in its manifest that it needs a **normal permission**, the **system automatically** grants the app that permission at install time.
- The system doesn't prompt the user to grant normal permissions, and users cannot revoke these permissions.

# Application Permission Protection Level

2. **Signature permissions:**

- The system grants these app permissions at install time, but only when the app that attempts to use a permission is **signed by the same certificate** as the app that defines the permission.
- It's granted automatically by the system if both applications are signed with the same certificate
- Both applications have been developed by the same company, this means that they most certainly will share the same application signature (**are signed with the same .keystore)**;
- So, we can take this into advantage and define a custom permission with this increased level of security — signature.

# Application Permission Protection Level

**2. Signature permissions (Continue..):**

- Advantages of this:
  - There's no need to prompt the user asking for a different type of permission in order to communicate with another application.
  - Since they share the same signature, the OS automatically grants this access; it's expected that since the company behind them is the same they're trustworthy.
  - Since this is addressed by the OS, there's no need to implement a validation mechanism.

# Application Permission Protection Level

**3. Dangerous Permissions (Continue…):**

- Dangerous permissions cover areas where the app wants **data or resources that involve the user's private information**, or could potentially affect the user's stored data or the operation of other apps.
- For example, the ability to read the **user's contacts** is a dangerous permission.
- android.permission_group.CALENDAR
  - android.permission.READ_CALENDAR
  - android.permission.WRITE_CALENDAR
- android.permission_group.CAMERA
  - android.permission.CAMERA
- android.permission_group.LOCATION
  - android.permission.ACCESS_FINE_LOCATION
  - android.permission.ACCESS_COARSE_LOCATION

# Application Permission Protection Level

**3. Dangerous Permissions (Continue…):**

- android.permission_group.CONTACTS
  - android.permission.READ_CONTACTS
  - android.permission.WRITE_CONTACTS
  - android.permission.GET_ACCOUNTS
- android.permission_group.MICROPHONE
  - android.permission.RECORD_AUDIO
- android.permission_group.SENSORS
  - android.permission.BODY_SENSORS
- android.permission_group.PHONE
  - android.permission.READ_PHONE_STATE
  - android.permission.READ_CALL_LOG
  - android.permission.WRITE_CALL_LOG
  - android.permission.ADD_VOICEMAIL

# Application Permission Protection Level

**3. Dangerous Permissions (Continue…):**

- android.permission_group.SMS
  - android.permission.SEND_SMS
  - android.permission.RECEIVE_SMS
  - android.permission.READ_SMS
  - android.permission.RECEIVE_WAP_PUSH
  - android.permission.RECEIVE_MMS
- android.permission_group.STORAGE
  - android.permission.READ_EXTERNAL_STORAGE
  - android.permission.WRITE_EXTERNAL_STORAGE

# Application Permission Protection Level

**3. Dangerous Permissions (Continue…):**

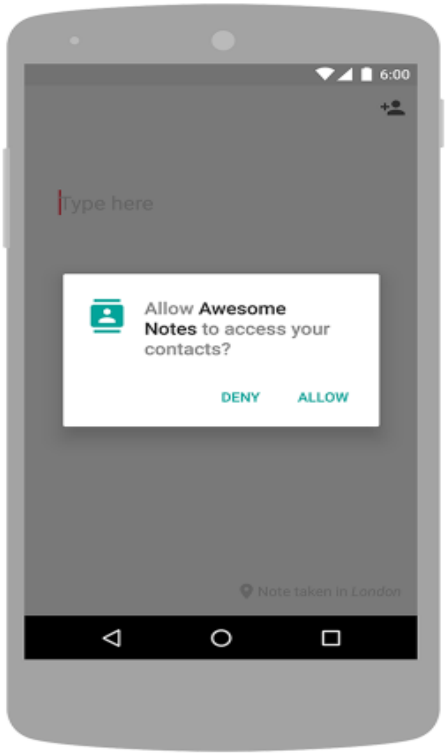
- If an app declares that it needs a dangerous permission, the user has to **explicitly grant the permission** to the app.
- Until the user approves the permission, your app cannot provide functionality that depends on that permission.
- To use a dangerous permission, your app must **prompt the user to grant permission at runtime**.

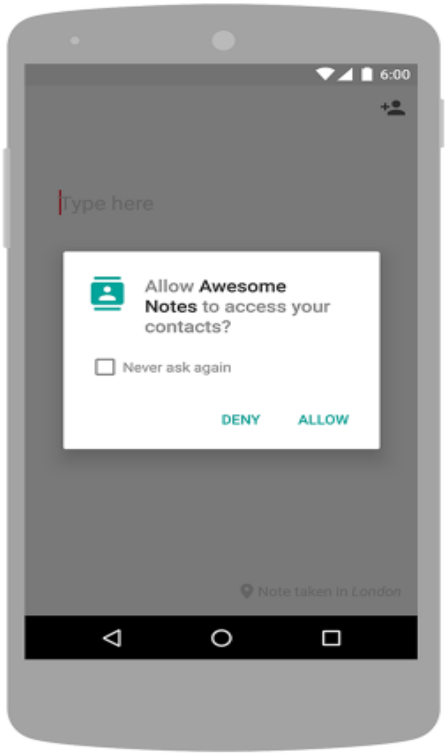
# Application Permission – Runtime Request

- If the device is **running Android 6.0 (API level 23)** or higher, and the app's **targetSdkVersion is 23** or higher, the user isn't notified of any app permissions at install time.
- Your app must ask the user to g**rant the dangerous permissions at runtime.**
- When your app requests permission, the user sees a system dialog (as shown in figure 1, left) telling the user which permission group your app is trying to access. The dialog includes a Deny and Allow button.

# Application Permission – Runtime Request

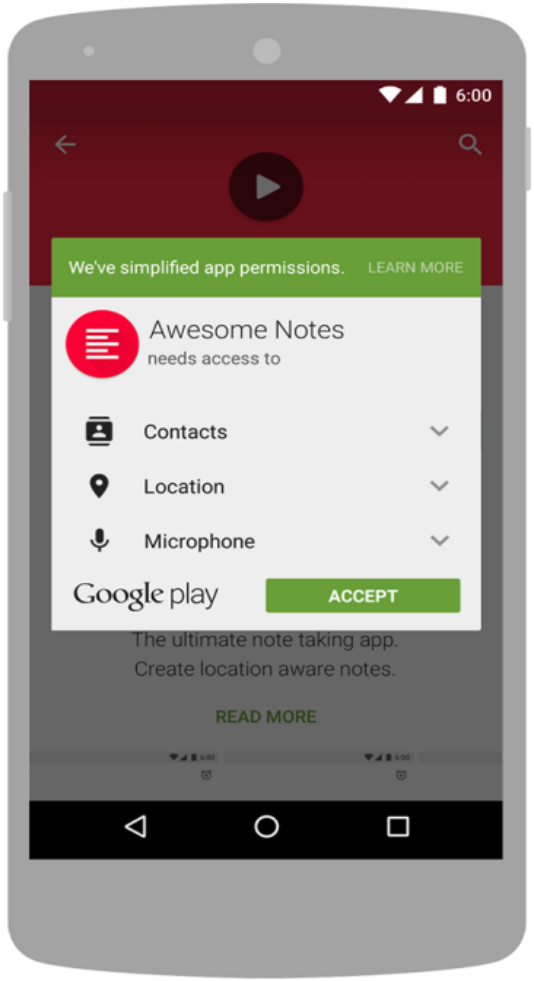
- ✓ The dialog includes a **Deny and Allow button.**
- ✓ If the user denies the permission request, the next time your app requests the permission, the dialog contains a checkbox that, when checked, indicates the user doesn't want to be prompted for the permission again



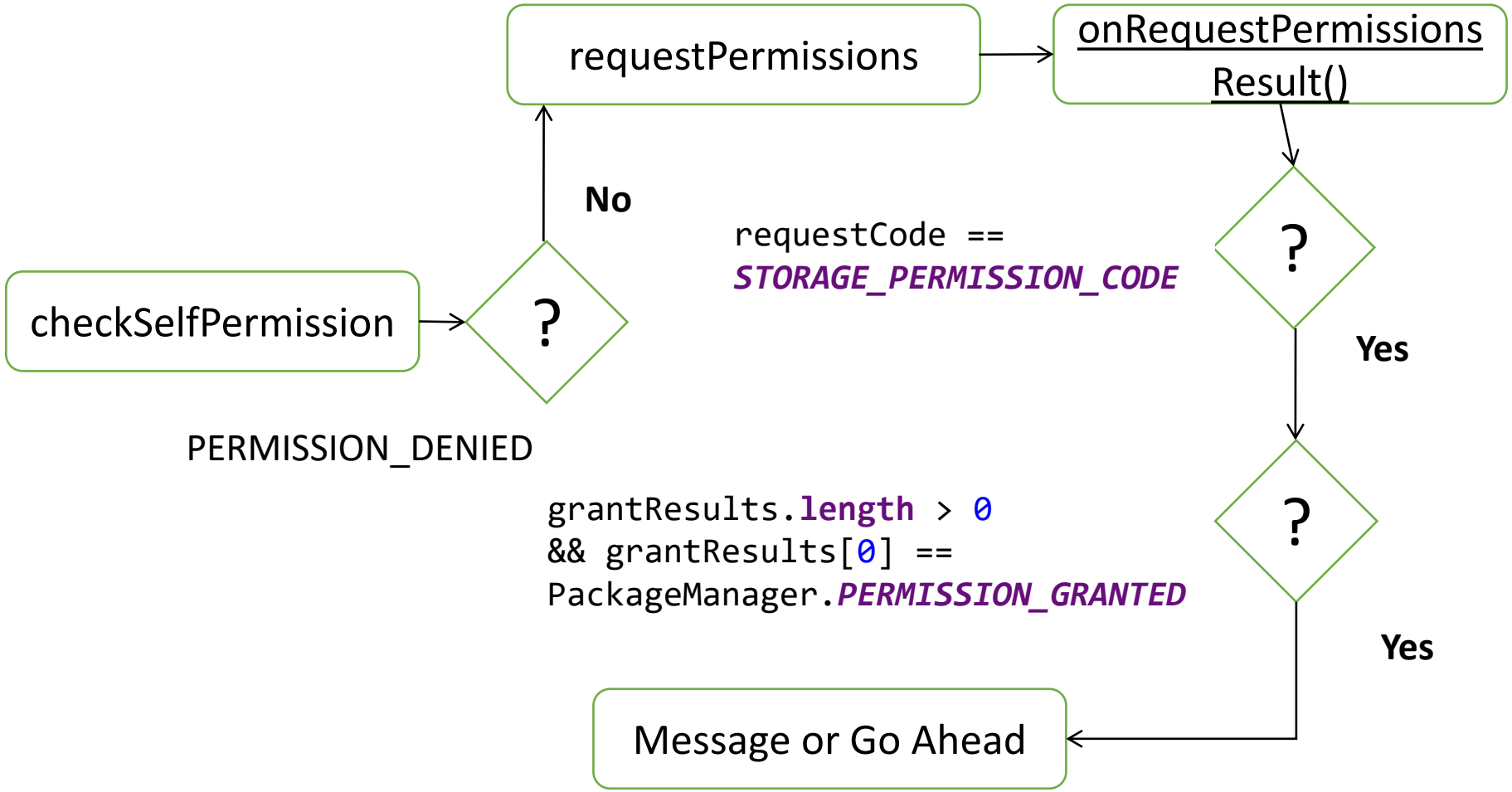



# Application Permission – Install time Request

- If the device is running **Android 5.1.1 (API level 22)** or lower, or the app's **targetSdkVersion is 22 or lower** while running on any version of Android, the system automatically asks the user to **grant all dangerous permissions for your app at install-time**.
- If the user clicks **Accept**, all permissions the app requests are granted. If the user denies the permissions request, the system cancels the installation of the app.
- If an app **update includes** the need for additional permissions the user is prompted to accept those new permissions before updating the app.



# Application Permission – Steps



# Application Permission – Steps

**Step-1**

- Declare the permission in Android Manifest file: In Android permissions are declared in AndroidManifest.xml file using the uses-permission tag.
- **<uses-permission android:name="android.permission.READ_CONTACTS" />**
- Here we are declaring contacts permission.

**Step – 2** Check whether permission is already granted or not.

- If permission isn’t already granted, request user for the permission:
- In order to use any service or feature, the permissions are required.
- Hence we have to ensure that the permissions are given for that. If not, then the permissions are requested.

# Application Permission – Steps

```
if(ContextCompat.checkSelfPermission(thisActivity,
Manifest.permission.READ_CONTACTS)
    != PackageManager.PERMISSION_GRANTED)
    { // Permission is not granted }
```

**Step-3** Request Permissions:

- When PERMISSION_DENIED is returned from the checkSelfPermission() method in the above syntax, we need to prompt the user for that permission.
- Android provides several methods that can be used to request permission, such **as requestPermissions().**

```
ActivityCompat.requestPermissions(MainActivity.this,
permissionArray, requestCode);
```

# Application Permission – Steps

- **Step – 4 Override onRequestPermissionsResult() method:**
  - onRequestPermissionsResult() is called when user grant or decline the permission.
  - RequestCode is one of the parameteres of this function which is used to check user action for corresponding request.
  - Here a toast message is shown indicating the permission and user action.

```
onRequestPermissionsResult(int requestCode, String[]
permissions,  int[] grantResults)
```

# Application Permission

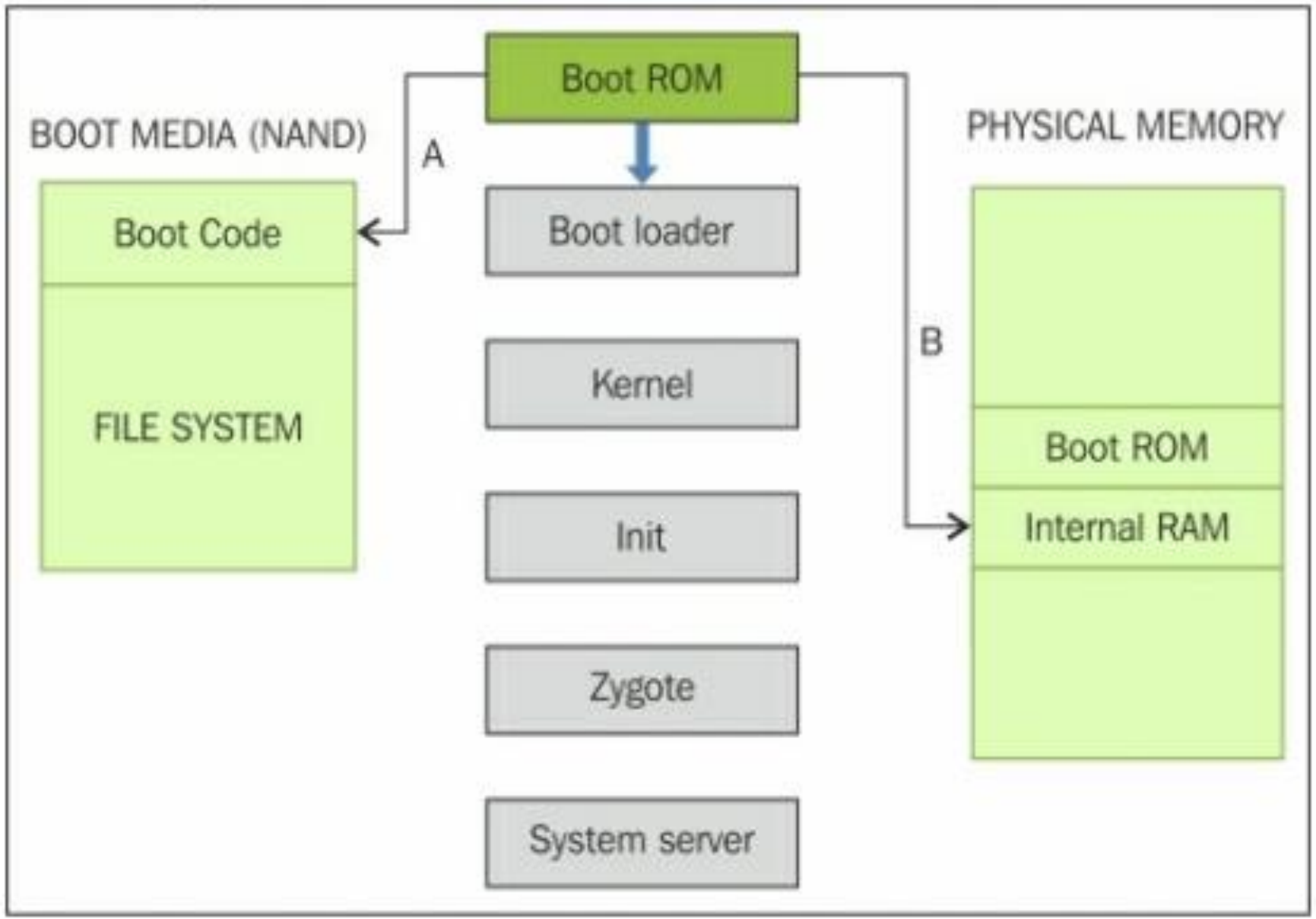
activity_main.xml | AndroidManifest.xml | MainActivity.kt

```
import ...
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        if(ActivityCompat.checkSelfPermission( context: this,Manifest.permission.READ_CONTACTS)!=PackageManager.PERMISSION_GRANTED)
        {
            ActivityCompat.requestPermissions( activity: this, arrayOf(Manifest.permission.READ_CONTACTS),  requestCode: 101)
        }else{
            accessApp()
        }
    }
    private fun accessApp() {
        Toast.makeText(applicationContext, text: "Congrats! You can access the App",Toast.LENGTH_LONG).show()
    }
    override fun onRequestPermissionsResult(
        requestCode: Int,
        permissions: Array<out String>,
        grantResults: IntArray
    ) {
        super.onRequestPermissionsResult(requestCode, permissions, grantResults)
        if(requestCode==101 && grantResults[0] == PackageManager.PERMISSION_GRANTED){
            accessApp()
        }else if(requestCode==101 && grantResults[0] == PackageManager.PERMISSION_DENIED){
            Toast.makeText(applicationContext, text: "Plz, allow to accesss the app",Toast.LENGTH_LONG).show()
        }
    }
```

# Android Boot Process

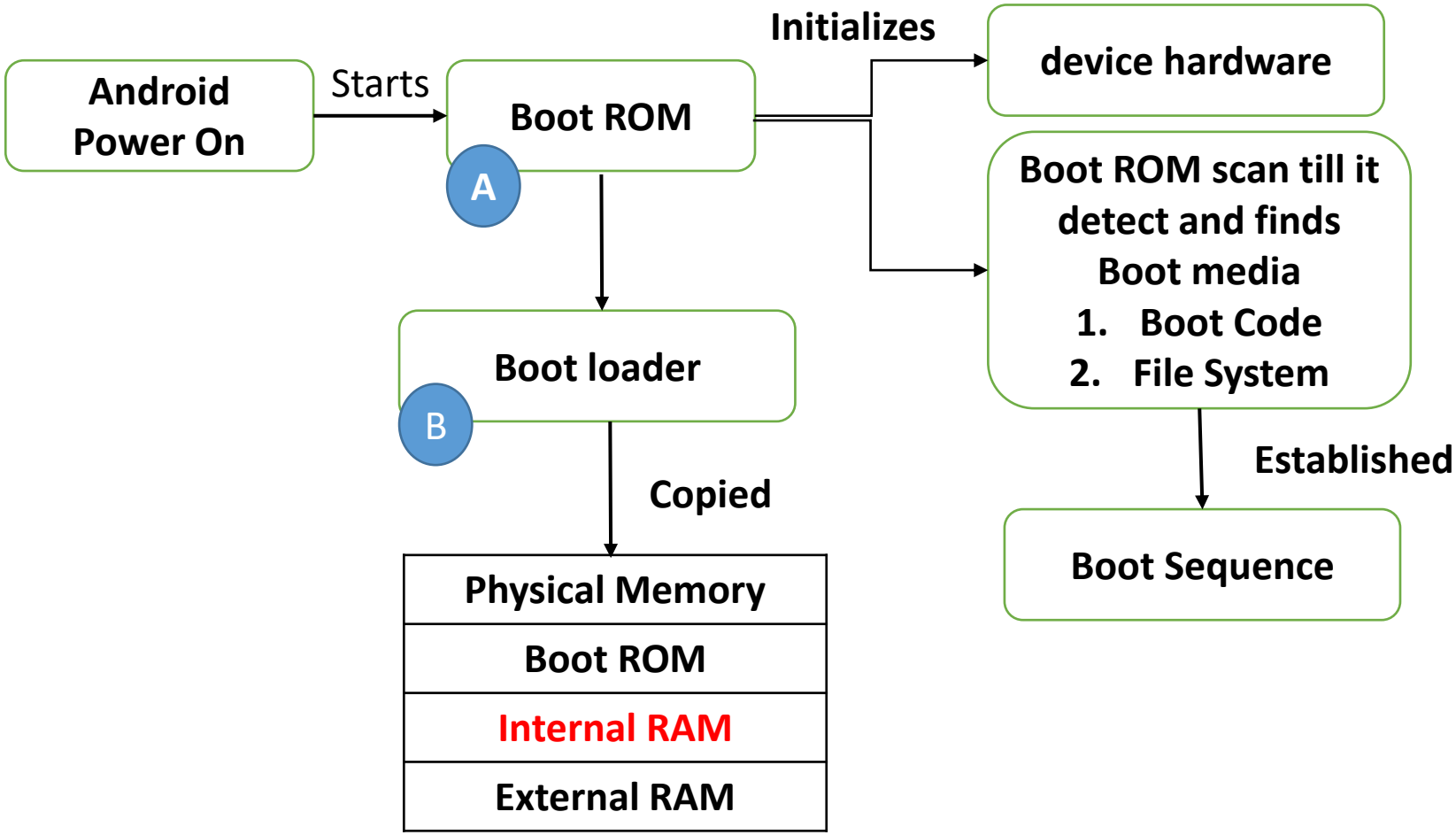
- When an Android device is first powered on, there is a sequence of steps that are executed, helping the device to load necessary firmware, OS, application data, and so on into memory.
- The sequence of steps involved in Android boot process is as follows:
  1. Boot ROM code execution
  2. The Boot loader
  3. The Linux kernel
  4. The init process
  5. Zygote and Dalvik
  6. The system server

# Android Boot Process




Back

# Android Boot Process – Boot Rom



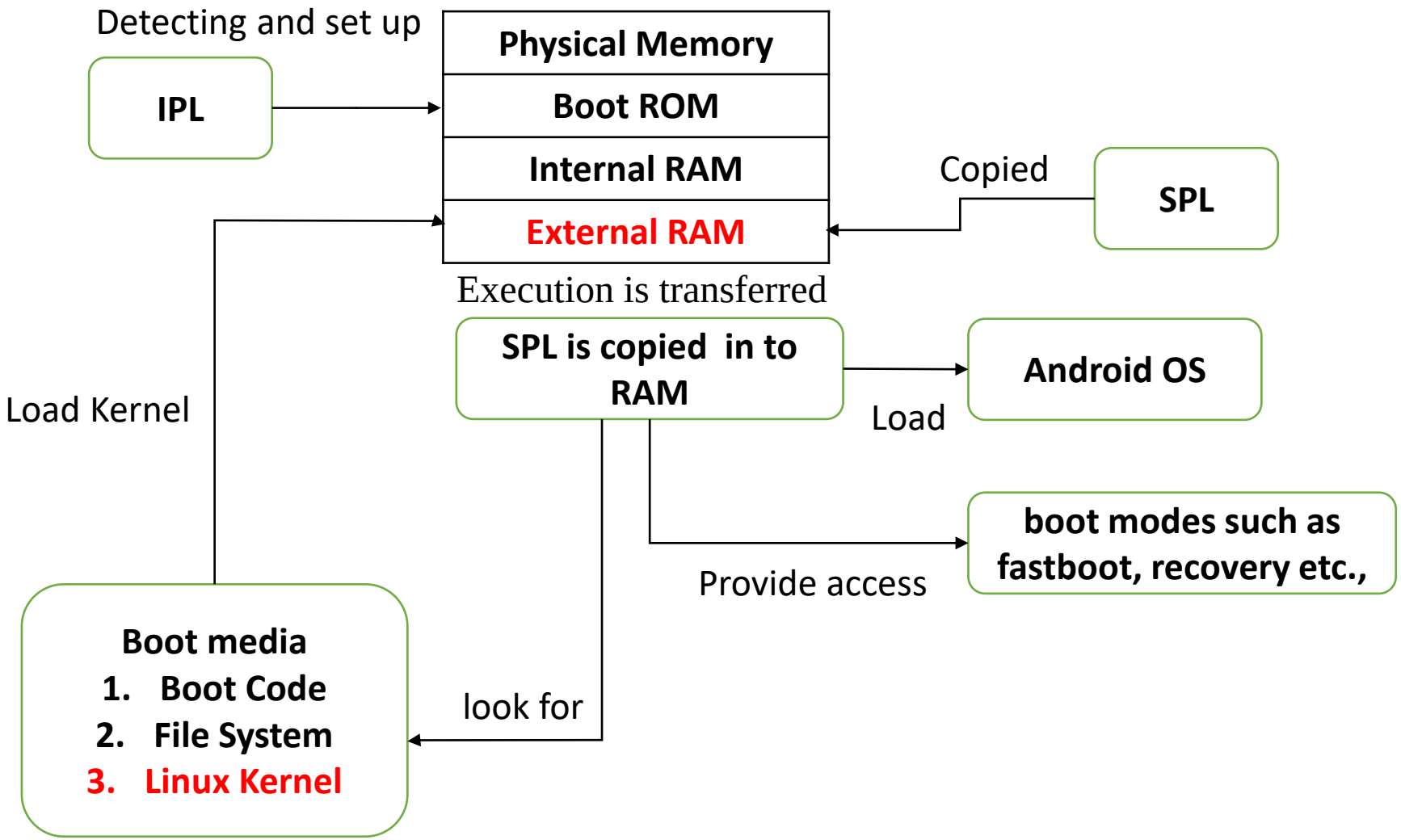

execution shifts to the code loaded into the RAM.

Back

# Android Boot Process – Boot Loader

- The boot loader is a piece of program that is executed before the operating system starts to function.
- There are two stages
  - Initial Program Load (IPL)
  - Second Program Load (SPL)
- In the first stage, it detects external RAM and loads a program which helps in the second stage.
- In the second stage, the bootloader setups the network, memory etc which requires to run Kernel.

# Android Boot Process – Boot Loader




Back

# Android Boot Process – Linux Kernel

- The Linux kernel is the heart of the Android operating system and is responsible for **process management, memory management,** and enforcing security on the device.
- After the kernel is loaded,
- It mounts the root file system (rootfs) and provides access to system and user data.
- When the memory management units and **caches have been initialized**, the system can use virtual memory and launch user space processes.
- The kernel will look in the **rootfs for the init process** and launch it as the **initial user space process.**

Back

# Android Boot Process – The init process

- The **init** is the very **first process** that starts and is the root process of all other processes - (**init.rc**).
- The init process will **parse** the **init.rc** script and launch the system service processes.
- At this stage, you will see the **Android logo** on the device screen.

Back

# Android Boot Process – Zygote and Dalvik

- Zygote is one of the first init processes created after the device boots.
- It initializes the Dalvik virtual machine and tries to create multiple instances to support each android process.
- Zygote facilitates using a shared code across the VM, thus helping to save the memory and reduce the burden on the system.

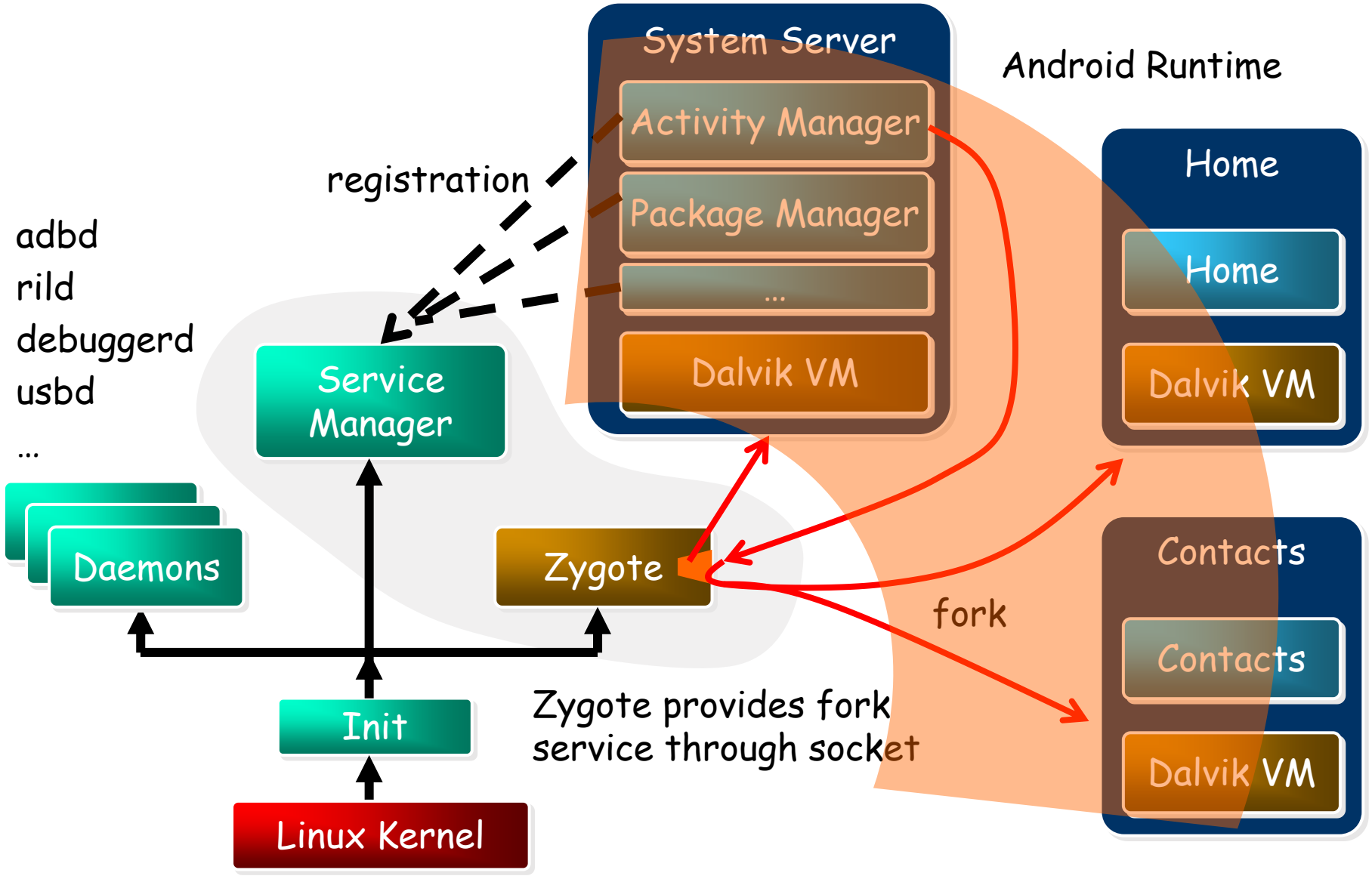
Back

# Android Boot Process – System Server

- All the core features of the device such as telephony, network, and other important functions, are started by the system server
- The following core services are started in this process:
  - Start Power Manager
  - Create Activity Manager
  - Start Telephony Registry
  - Start Package Manager
  - Set Activity Manager Service as System Process
  - Start Context Manager etc.,

Back

# Android Boot Sequence



# Android Partitions

- The Android partition layout varies between vendors and versions.
- However, a few partitions are present in all the Android devices.
- Partitions:
  - Boot Loader
  - Boot
  - Recovery
  - UserData
  - System
  - Cache
  - Radio

# Android Partitions

- Boot Loader :
  - This partition stores the **phone's boot loader program**.
  - This program takes care of initializing the **low level hardware when the phone boots**.
  - Thus, it is responsible for booting the Android kernel and booting into **other boot modes**, such as the recovery mode, download mode, and so on.

# Android Partitions

- Boot
  - As the name suggests, this partition has the information and files required for the **phone to boot.**
  - It contains the **kernel and RAM disk.** So, without this partition, the phone cannot start its processes.
- Recovery
  - Recovery partition allows the device to boot into the **recovery console** through which activities such as phone **updates and other maintenance operations are performed.**
  - For this purpose, a minimal Android boot image is stored.

# Android Partitions

- Userdata
  - This partition is usually called the **data partition** and is the **device's internal storage for application** data.
  - A bulk of user data is stored here, and this is where most of **our forensic evidence will reside**.
  - It stores all app data and standard communications as well.
- System
  - All the major components other than **kernel and RAM disk** are present here.
  - The Android system image here contains the Android **framework, libraries, system binaries, and preinstalled** applications.
  - Without this partition, the device cannot boot into normal mode.

# Android Partitions

- Cache
  - This partition is used to store **frequently accessed data** and various other files, such as recovery logs
  - and update packages downloaded over the cellular network.
- Radio
  - Devices with **telephony capabilities have a baseband image stored in** this partition that takes care of various telephony activities.

# Android Partitions - Practical

- adb shell
- root@android: cat proc/partitions
  - Using **cat /proc/partitions** you will get the actual partition name identifier and the number of blocks in the partition
- root@android: cat /proc/mounts
  - Using **cat /proc/mounts** shows the partition file path, the alias, the filesystem type, the starting inode, the number of blocks, read/write status (and other parameters of the individual partition that I'm not entirely sure of).
- root@android: df
  - Using df lists the filesystem path alias and size info (total size, used, free and block size)

# Android File Systems

- Filesystem refers to the way data is stored, organized, and retrieved from a volume.
- A basic installation may be based on one volume split into several partitions; here, each partition can be managed by a different filesystem.
- **Microsoft Windows** users are mostly familiar with the FAT32 or NTFS filesystem, whereas **Linux** users are more familiar with the EXT2 or EXT4 filesystem.
- As is true in Linux, Android also utilizes mount points and not drives (that is C : or E :). Each filesystem defines its own rules to manage the files the volume.
- Depending on these rules, each filesystem offers a different speed for file retrieval, security, size, and so on.

# Android File Systems

- Understanding the filesystem is very important in Android forensics, as it helps us gain knowledge of how the data is stored and retrieved.
- This knowledge about properties and the structure of a filesystem will prove to be useful during forensic analysis.
- In Linux, mounting is an act of attaching an additional filesystem to the currently accessible filesystem of a computer.
- Linux is known to support a large number of filesystems.
- These filesystems used by the system are not accessed by drive names, but instead are combined into a single hierarchical tree structure that represents these filesystems as a single entity.

# Android File Systems

- The filesystems are mounted on to a directory, and files present in this filesystem are now the contents of that directory.
- This directory is called a mount point.
- It makes no difference whether the filesystem exists on the local device or on a remote device. Everything is integrated into a single file hierarchy that begins with root.
- Each filesystem has a separate kernel module that registers the operations that it supports with something called virtual file system (VFS).
- VFS allows different applications to access different filesystems in a uniform way.

# Android File Systems - Practical

- The filesystems supported by the Android kernel can be determined by checking the contents of the filesystems file that are present in the proc folder.
- root@android: cat  /proc/filesystems
- The filesystems preceded by the nodev property are not mounted on the device.

# Android File Systems

- The filesystems present in Android can be divided into three main categories, which are as follows:
  - Flash memory filesystems
  - Media-based filesystems
  - Pseudo filesystems

# Android File Systems – Flash Memory File Systems

- ## Flash Memory File Systems
- Flash memory is a type of non-volatile memory that can be erased and reprogrammed in units of memory called blocks.
- Due to the particular characteristics of flash memories, special filesystems are needed to write over the media and deal with the long erase times of certain blocks.
- While the supported filesystems vary on different Android devices, the common flash memory filesystems are as follows:
    1. Extended File Allocation Table (exFAT)
    2. Flash Friendly File System (F2FS )
    3. Journal Flash File System version 2 (JFFS2)
    4. Yet Another Flash File System version 2 (YAFFS2)
    5. Robust File System (RFS):

# Android File Systems – Flash Memory File Systems

1. Extended File Allocation Table (**exFAT**):
   - This type of filesystem is a Microsoft proprietary. filesystem optimized for flash drives.
2. Flash Friendly File System (**F2FS** ):
   - This type of filesystem is introduced by Samsung as an open source filesystem.
3. Journal Flash File System version 2 (**JFFS2**):
   - Support for flash devices.
   - Better performance. JFFS treated the disk as a purely circular log. This generated a great deal of unnecessary I/O. The garbage collection algorithm in JFFS2 makes this mostly unnecessary.
4. Yet Another Flash File System version 2 (**YAFFS2**): Currently, YAFFS2 is not supported in newer kernel versions, but certain mobile manufacturers might still continue to support it.
5. Robust File System (**RFS**):
   - RFS is a Samsung file system for user space applications formatted as RFS (Robust File System).
   - RFS is a FAT 16/32 based file system with a sort of journalling system on it.
   - The code to build and use the RFS file system is proprietary and only used in Samsung equipment.

# Android File Systems – Media Based File Systems

- Besides the flash memory filesystems discussed earlier, Android devices typically support the following media-based filesystems:
  - EXTended file system (EXT2/EXT3/EXT4)
  - File Allocation Table (FAT)
  - Virtual File Allocation Table (VFAT)

# Android File Systems – Media Based File Systems

- EXTended file system (EXT2/EXT3/EXT4)
  - This was one of the first filesystems and used the virtual filesystem.
  - EXT2, EXT3, and EXT4 are the subsequent versions. Journaling is the main advantage of EXT3 over EXT2.
  - With EXT3, in the case of an unexpected shutdown, there is no need to verify the filesystem.
  - The EXT4 filesystem, the fourth extended filesystem, has gained significance with mobile devices that implement dual-core processors.

# Android File Systems – Media Based File Systems

- File Allocation Table (FAT)
  - These filesystems, such as FAT12, FAT16, and FAT32, are supported by the MSDOS driver.
- Virtual File Allocation Table (VFAT)
  - This filesystem is an extension of the FAT16 and FAT32 filesystems.
  - Microsoft's FAT32 filesystem is supported by most Android devices.
  - It is supported by almost all the major operating systems, including Windows, Linux, and Mac OS.
  - This enables these systems to easily read, modify, and delete the files present on the FAT32 portion of the Android device.
  - Most of the external SD cards are formatted using the FAT32.

# Android File Systems – Pseudo File Systems

- Here are pseudo filesystems that can be thought of as logical groupings of files.
  - control group (cgroup)
  - rootfs
  - procfs
  - Sysfs
  - tmpfs: