ECE 420
Lecture 11
November 11 2019

Origins of Android

- Founded in 2003, with the intention of making “smarter mobile devices that are more aware of its owner’s location and preferences”
- Initial application was intended as an operating system for digital cameras
- Acquired by Google in 2005 and retargeted at the mobile phone market
  - Linux-based system allowed the OS to be distributed for free to manufacturers
  - Google to make money via apps and services
- Android was still a secret when Apple announced the iPhone in 2007

Origins of Android

- Late 2007, Google started the Open Handset Alliance among various phone manufacturers
  - Established hardware standards upon which Android could run
- First official Android based smartphone was T-Mobile G1 released in 2008
- Android lines popularized by the ‘Droid’ line from Motorola

Android Codenames

- Initial versions of the OS did not have official codenames
- Android version 1.5 was named ‘Cupcake’ starting the trend of alphabetically organized sweets and desserts
- No official reason has been given behind this naming decision
- Google actually unveils a code-name themed statue at their headquarters with each release
- Sadly, the naming trend has stopped with Android version 10 (perhaps no good options for ‘Q’?)

API Level Codenames

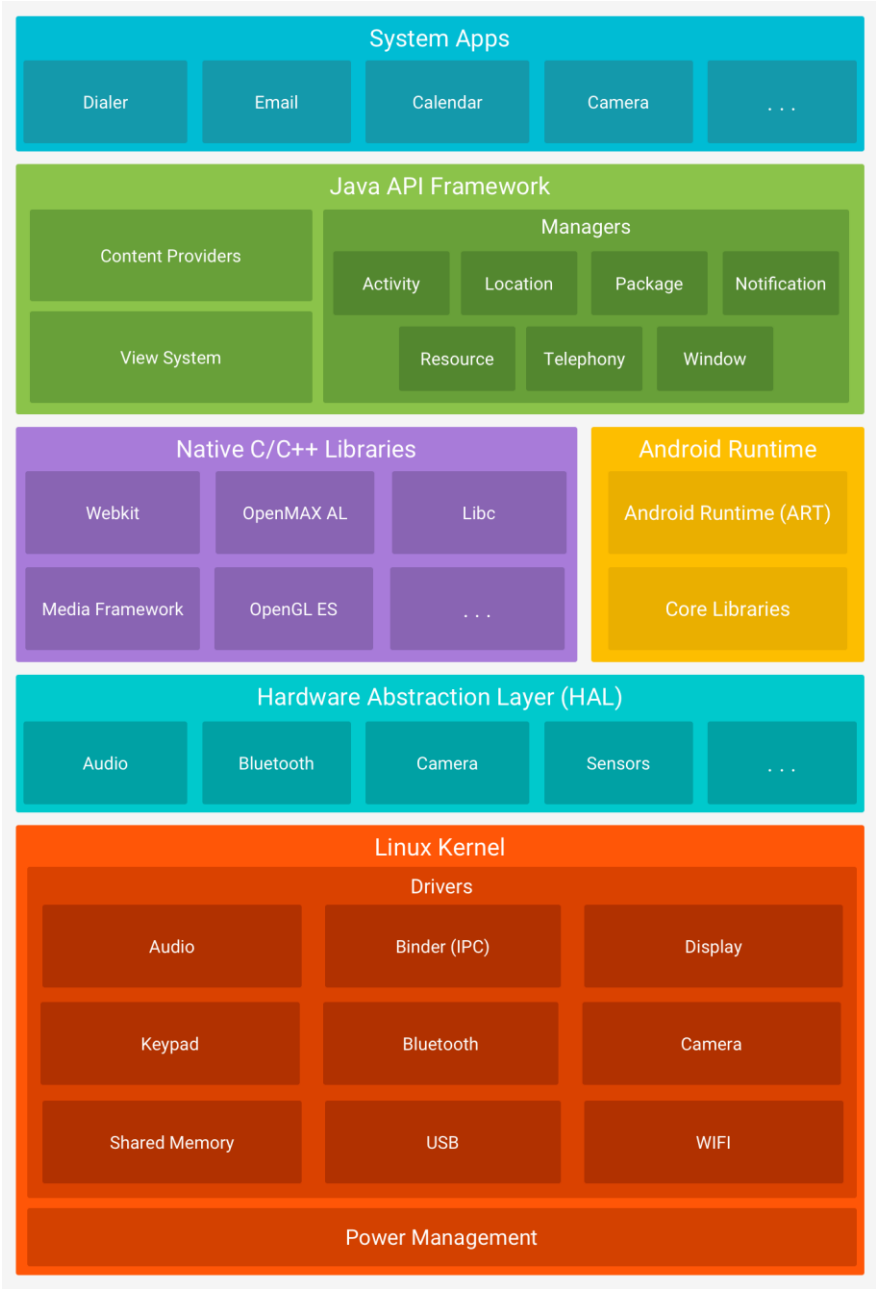
Code Name	API Level
B.C.	1, 2
Cupcake	3
Donut	4
Eclair	5, 6, 7
Froyo	8
Gingerbread	9, 10
Honeycomb	11, 12, 13
Ice Cream Sandwich	14, 15

Code Name	API Level
Jelly Bean	16, 17, 18
KitKat	19, 20
Lollipop	21, 22
Marshmallow	23
Nougat	24, 25
Oreo	26, 27
Pie	28
‘10’	29

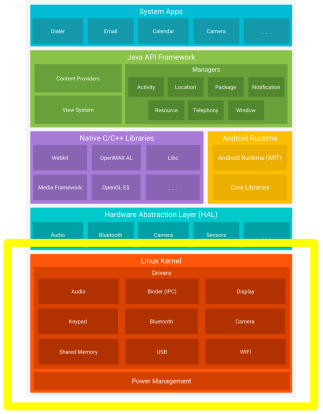
Android Motivations

- Secure and efficient platform
- Applicable for a broad range of devices, with significant resource constraints
- Customizable, low cost option for manufacturers
- Platform for Google to embed and monetize services

Android Software Stack

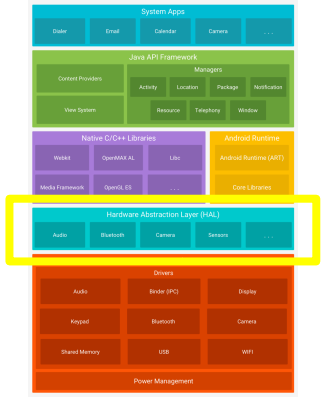


Linux Kernel



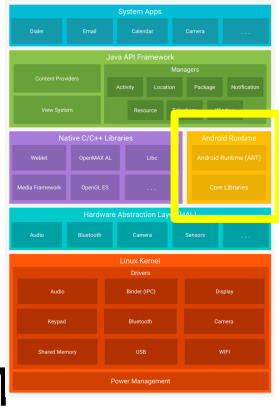
- The foundation of the Android platform is the Linux kernel
- Upper layers of the software stack utilize underlying functionalities such as threading and low-level memory management
- The Linux kernel as a common base facilitates hardware driver development for device manufacturers
- Linux also provides several security features
  - User-based permissions
  - Process isolation
  - Secure interprocess communication
  - Modularized, with ability to remove unneeded parts of the kernel

Hardware Abstraction Layer



- Provides standard interfaces for device hardware
  - Abstracts details of interactions with device layer modules
- Consists of library modules as interfaces for each type of hardware component (Bluetooth, camera, etc.)
- Dynamically loaded by Android system when hardware is accessed

Dalvik Runtime



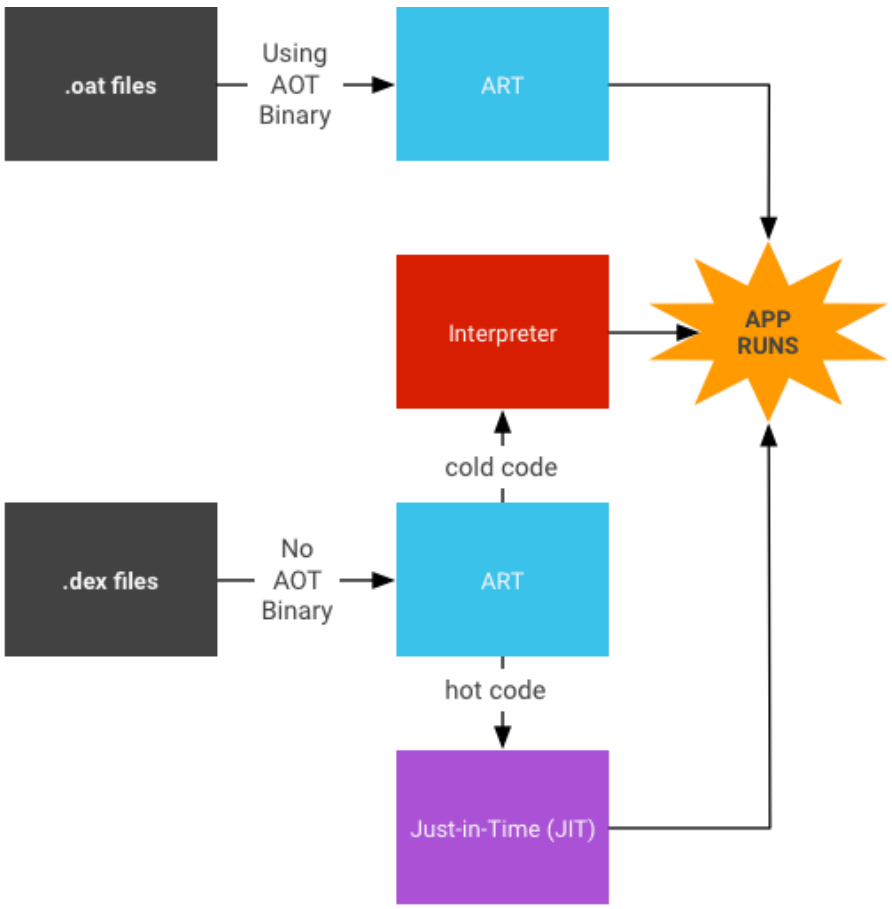
- Dalvik is the virtual machine that executes Android applications
- Aimed at running processes efficiently
  - Register-based instead of stack-based like the Java VM
  - Minimize memory footprint of applications, VM
- Conversion from Java bytecode to Davlik bytecode (.dex) can take advantage of more complex instruction set to save space, along with other optimizations
- Additional optimization can be performed at install time
  - Inlining of libraries, byte order swapping
- Discontinued after Android 4.4

Android Runtime (ART)

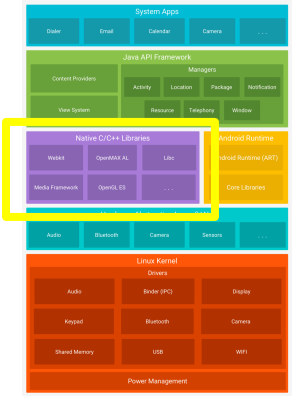
- Replacement for the Dalvik VM, runs the same .dex bytecode
- Faster interpreter, faster native methods
- Ahead-of-time compilation
  - Compile the full application to native code
- Improved garbage collection
  - Parallelized GC
  - Special handling for short-lived objects
  - Compacting GC for reduction of memory fragmentation
- Better debugging support, including a dedicated sampling profiler, detailed diagnostic exceptions and crash reporting, and the ability to set watchpoints
- More aggressive optimizations for loops and inlining

AOT vs. JIT

- AOT has the benefits of off-line optimization, fast app startup, no overhead from VM manager
- JIT has the benefits of being able to take code behavior into account for optimizations
- Current Android does both
  - Run AOT when available
  - If no AOT code, use JIT and collect profiling stats
  - Periodically use profiling info to drive full AOT compile
- "Continuously improve applications as they run"

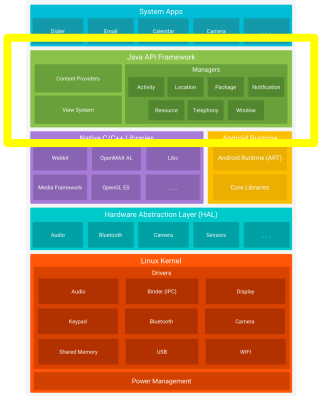


Native C++ Libraries



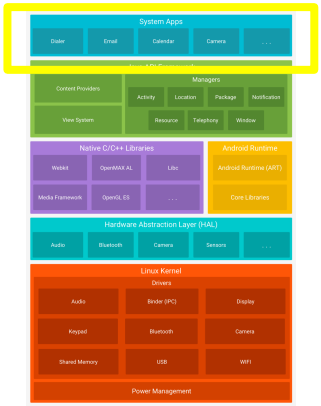
- Many core Android system components and services are built from native code
  - They require native libraries written in C and C++
- The Java framework APIs to expose the functionality of some of these native libraries to apps (e.g. OpenGL for graphics support)
- Android NDK (Native Development Kit) allows developing an app in C or C++ code and can access these native platform libraries directly

Java API Framework



- The Java API framework exposes the entire feature-set of system components and services of the Android OS'
  - The View System is used to build an app's UI, including lists, grids, text boxes, buttons
  - The Resource Manager provides access to non-code resources such as strings, graphics, and layout files
  - The Notification Manager enables all apps to display custom alerts in the status bar
  - The Activity Manager manages the lifecycle of apps and provides a "back stack" framework
  - Content Providers enable apps to access data from other apps, such as the Contacts app, or to share their own data

System Apps



- Android comes with a set of core apps
  - Email, SMS messaging, calendars, internet browsing, contacts, etc.
- Initial core apps no special status and can be replaced by a third-party app as the default SMS messenger, or browser, etc.
- System apps provide functionality as apps for the end user and to also capabilities usable by other apps
  - Example: Delivering an SMS message involves invoking the SMS app

System Level Protections

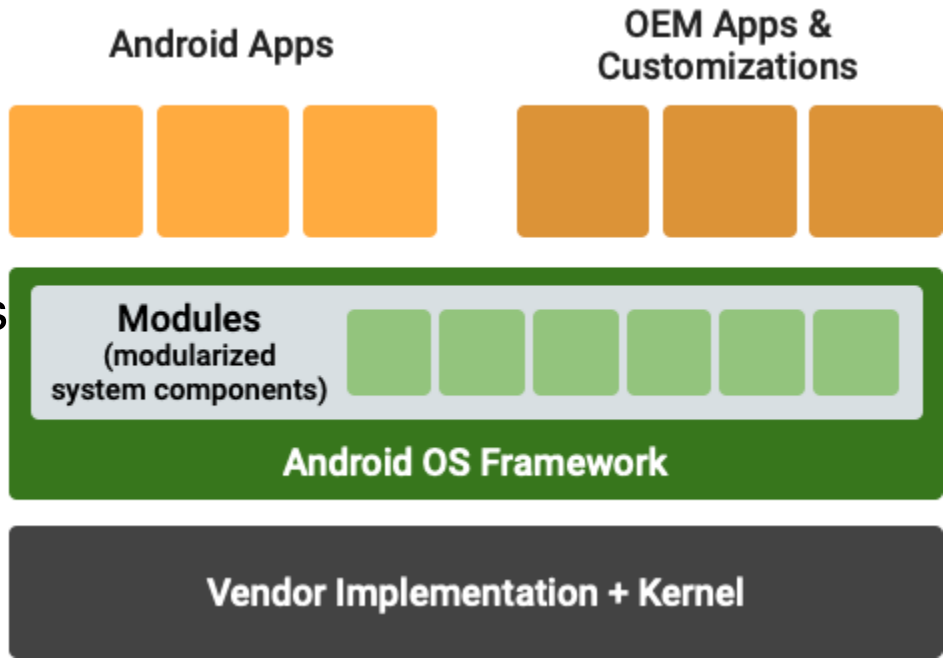
- Android leverages Linux user-based protection to identify and isolate apps and app resources
  - Each app is assigned a unique user ID and run in its own process
- The UID is also used to set up a kernel-level Application Sandbox which enforces security between apps and the system (limited OS access) and other apps
- Kernel level sandboxing allows native code and interpreted code to be comparably secured
- Each app explicitly requests permissions to access resources (Internet, camera, GPS, etc.) at install time

Interprocess Communication

- Processes can communicate using any of the traditional UNIX-type mechanisms (files, sockets, signals) but Linux permissions still apply
- Android specific IPC (recommended):
  - Binder: A lightweight remote procedure call mechanism designed for high performance, implemented in the Linux driver
  - Intents: A simple message object that represents an "intention" to do something. For example, to display a web page, the app expresses its "Intent" to view the URL by handing off an intent object to the system. The system passes it to the handler and runs it.
  - ContentProviders: A storehouse that provides access to data on the device. Apps can access data other apps have exposed, and can define its own Content Providers to share data

Modularization

- Android 10 continues modularization efforts into the system components
  - Enables system components to be updated with critical bug fixes and other improvements as needed
- Module updates don't introduce new APIs, so will not break existing functionality



- Module package installs/rollbacks are performed atomically, so all updated successfully or none are in the case of an error

System Updates

- Modern Android devices have two partitions, the current and backup partition
- Received updates are applied to the backup partition
- Upon next reboot, the system will swap partition roles
  - Backup partition becomes active, current partition becomes backup
- If an error occurs and the system is not bootable, the system is rolled back to the backup partition
  - Update can be attempted again

System Updates

- Prevents interfering with normal operation and moves upgrade to a background process
- ‘Cost’ of updating is only a system reboot
- System is resilient against errors introduced
- Updates can be streamed into the new partition instead of downloading then installing packages
  - No caching required
- Permits for system module updates outside of usual release cycle (continuous delivery) and independent from device manufacturer

Kotlin Background

- JetBrains unveiled Project Kotlin in July 2011
- Targeting the JVM, it aimed to contain “all the features they were looking for” that were not present in other languages
- Kotlin version 1.0 was released in February 2016.
- Google added Kotlin support in Android Studio in 2017, in addition to Java and C++
- Google announced Kotlin is the preferred language for Android development in May 2019
- The project name is derived from Kotlin Island near St. Petersburg.

Compatibility/Interop

- Interoperability with Java is a key concern of Kotlin
- Kotlin compilers can target JVM bytecode, allowing for easy mixing of Java and Kotlin based code
  - Leverage existing libraries and frameworks
  - Run on existing VM infrastructure
  - Progressive code migration
- Kotlin is also an open-source language

Kotlin Highlights

- Everything is an object in Kotlin – there are no primitive types
  - Methods can be called on any object
- Variables are defined as constant or mutable

```
val i: Int = 42
var j: Int = 42
```

- Classes are ‘final’ by default unless explicitly marked ‘open’
- Classes that only represent data marked as ‘data’
- ‘If’ constructs are expressions rather than statements

```
println(if(x > 10) "greater" else "smaller")
```

Kotlin Highlights

- Use of iterables in for loops

```
for (item in array) {
    print(item)
}
```

- ‘when’ construct for case selection (value and type)

```
when (x) {
    1, 2 -> print(“Target”)
    in 3..20 -> print(“Too low”)
    !in 21..40 -> print(“Too high”)
    else -> print(“Undefined”)
}
```

- Lambda expressions

```
var a: (Int) -> Int = { i: Int -> i * 2 }
```

Null Safety

- Missing NULL checks have been identified as a frequent source of errors
- Kotlin moves runtime errors to compile time errors
  - Improve program safety
  - Reduce the amount of defensive checks required
- Variables are not allowed to be null, unless explicitly declared as a nullable type
  - Append a '?' to the type, e.g. String?
- Compiler then can check for
  - Assignment of null to a non-nullable variable
  - Use of a nullable type without null checks

Null Safety

- Methods of a nullable object can be called if null check is performed

```
if (foo != null)
    check = foo.docheck()
else
    check = false
```

- The ‘safe call’ operator ?. can also be used. This returns a null value if the object is null. This can be chained together.

```
check = foo?.docheck()
```

- The Elvis operator ?: allows for default values to be returned in case of a null operand.

```
check = foo?.docheck() ?: false
```

Kotlin Summary

- Kotlin aims to provide lightweight mechanisms to improve programmer productivity and program safety
  - Concise program syntax
  - Null safety mechanisms
  - Compile time checking for common program errors
- Interoperability with existing Java runtime resources
- Supports all tooling inside of Android Studio (code inspection, etc.)
- Future plans for iOS compatibility

This Week

- Continued Final Project Work
- Milestone 1 Demo due in lab