CS 478: Software Development for Mobile Platforms

Ugo Buy
Department of Computer Science
University of Illinois at Chicago

January 15, 2019

Course learning objectives

1. Proficiency in developing Android apps
   - Learn and apply all app components: (1) Activities, (2) Broadcast receivers, (3) Content providers, and (4) Services
   - Sensors, location and maps

   We'll use the Android Studio IDE
2. Relate CS concepts to Android OS and Android app design

   Android OS uses concepts from many areas of CS
   - Design patterns, GUI design, OS design, Database design, OO PLs...
   - Explore Android design and source code
   - **Emphasis on design patterns**

Why Android?

Many reasons

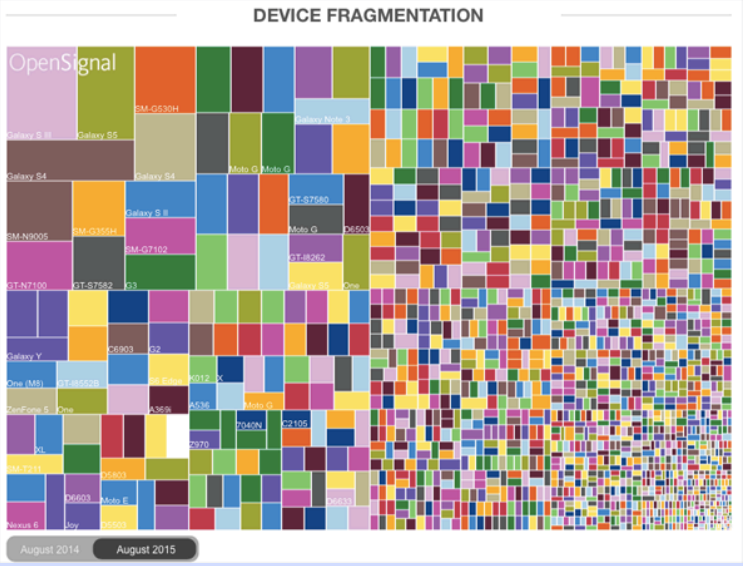
1. App development possible on all platforms
   - Android Studio IDE runs on MS Windows (7 and up), Mac OSX (10.10 and up), and Linux (GNOME and KDE)
   - Java 8 SDK
   - 8GB RAM recommended (can live with 4GB, but painful at times)
2. Open source OS
   - Can download source code for most versions
   - We'll look at source code on and off and explore design principles
3. CS Dept's Instructional Computing Lab (ICL) no longer has Macs
   - iOS apps can only be developed on Macs

The challenges of app development

1. Rapid evolution cycles of ecosystems
   - Typically, one or two major OS releases every year
   - New features in each release (e.g., Google Location Services)
   - Rapid obsolescence of development tools and techniques
   - Learning materials (e.g., books, online courses) typically obsolete as soon as released
2. Fragmentation of ecosystems
   - Plethora of hardware devices and software versions (e.g., different screen sizes, pixel densities, custom OSs, OS APIs)
   - iOS: Swift vs. Objective C; iPhone vs. iPad vs. iPod
   - Compatibility across different OS + language versions

Android fragmentation map, 2015

Source: http://fortune.com/2015/08/05/apple-android-fragmentation/



Key materials!

- Android:
  - Official on-line Android documentation, e.g.,
    http://developer.android.com/index.html
    https://developer.android.com/studio/
    http://developer.android.com/training/index.html
  - Adam Porter's excellent MOOC (Univ. of Maryland)

    https://www.coursera.org/learn/android-programming
  - No books needed for this course!

Course requirements and grading

The following are subject to minor changes:

- Quizzes: 2 Quizzes + 1 Homework (9% course grade)
  - Quiz on prerequisites next time (this Thursday)!
- Five projects (31% of course grade):
  - All individual
- Midterm exam (3/5/2019 ± 1 lecture, 25% of course grade)
- Final exam (Week 16: 10:30 am—12:30 pm on Monday 5/6/2019 – 35% of course grade)

Course prerequisites

- Must have: CS 342 (Software Design)
  - OO paradigm (classes, inheritance, polymorphism, abstract classes)
  - Java (classes, subclasses, interfaces, anonymous classes, class inheritance vs. interface inheritance, threads, locking mechanisms)
  - Design patterns (Gang of Four patterns: Composite, Adapter, Wrapper, Command, Iterator, etc.)
- Knowledge of some OS design principles desirable (e.g., from CS 361)
  - Familiarity with multithreading and thread synchronization
- Quiz on prerequisites next time

Course schedule (subject to changes)

- Week 1: Quiz 1 (on prerequisites)
- Week 2: Homework due
- Week 3: Project 1 due (Android basics)
- Week 4: Project 2 due (Activities and intents)
- Week 5: Quiz 2
- Week 7: Midterm exam (3/5/2019 ± 1 lecture)
- Week 9: Project 3 due (Fragments and background tasks)
- Week 11: Project 4 due (Broadcast receivers services)
- Week 14: Project 5 due (Files, databases and content providers)
- Week 16: Final exam (10:30 am—12:30 pm Monday 5/6/2019)

Background on Android

- OS designed for mobile devices with a touch screen display
  - Based on a Linux kernel
  - Developed by Google as an open source project
  - Most popular OS: Sales in 2012—17 exceeded those of iOS, Mac OS X and MS Windows **combined**
  - Over 2 Billion currently active users (Google estimated)
  - First released in 2007
  - First commercial product: HTC Dream (released Oct. 2008)
  - Typically integrated with proprietary code written by device vendors and with Google Mobile Services (no sources available ☹)

Versions and API levels

- Each major version of Android introduces new features and modifies existing ones
  - New features invoked programmatically through appropriate API calls
- Consequence: APIs change with each major release
  - New functions/classes added
  - Some functions/classes deprecated and eventually dropped
- Android Studio lets you choose target OS version (i.e., API level) for your apps
  - We'll target almost-latest version (Oreo– API level 275)
  - Test apps on different devices and OS versions with built-in emulator

Major version releases

- 1.0 (Alpha) – API level 1 – Sept. 2008
- 1.1 (Beta) – API level 2 – Feb. 2009
- 1.5 (Cupcake) – API level 3 – Apr. 2009
- 1.6 (Donut) – API level 4 – Sept. 2009
- 2.0, 2.1 (Éclair) – API levels 5, 6, 7 (Oct. 2009—Jan. 2010)
- 2.2 (Froyo) -- API level 8 (May 2010)
- 2.3 (Gingerbread) – API levels 9, 10 (Dec. 2010)
- 3.0, 3.1, 3.2 (Honeycomb) – API levels 11, 12, 13 (Feb–Jul. 2011)
- 4.0 (Ice Cream Sandwich) – API levels 14, 15 (Oct–Dec. 2011)
- 4.1—4.3 (Jelly Bean) – API levels 16, 17, 18 (Jul. 2012–Jul. 13)
- 4.4 (KitKat) – API levels 19 and 20 (Oct. 2013)
- 5.0, 5.1 (Lollypop) – API levels 21 and 22 (Nov. 2014)
- 6.0 (Marshmallow) – API level 23 (October 2015)
- 7.0, 7.1 (Nougat) – API levels 24 and 25 (August 2016)
- 8.0, 8.1 (Oreo) – API level 26, 27 (August 21, 2017)
- 9.0 (Pie) – API level 28 (August 6, 2018)

Source: https://en.wikipedia.org/wiki/Android_version_history

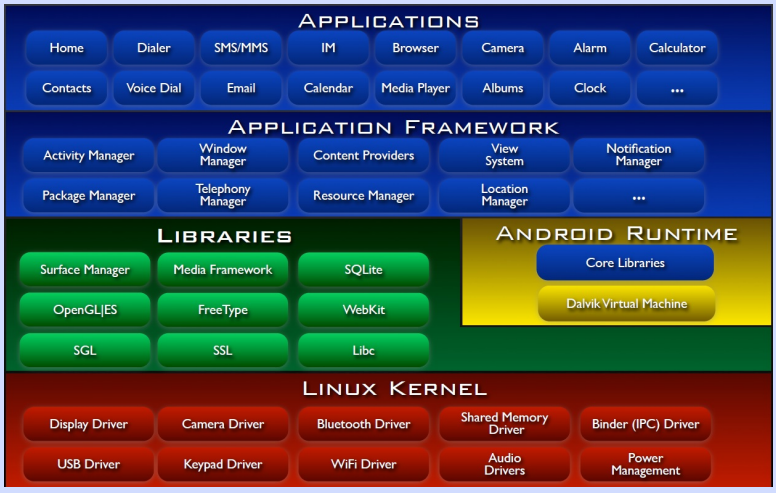
Use by version (Jan. 2019)

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.2%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.3%
4.1.x	Jelly Bean	16	1.1%
4.2.x		17	1.5%
4.3		18	0.4%
4.4	KitKat	19	7.6%
5.0	Lollipop	21	3.5%
5.1		22	14.4%
6.0	Marshmallow	23	21.3%
7.0	Nougat	24	18.1%
7.1		25	10.1%
8.0	Oreo	26	14.0%
8.1		27	7.5%

Source: https://developer.android.com/about/dashboards/

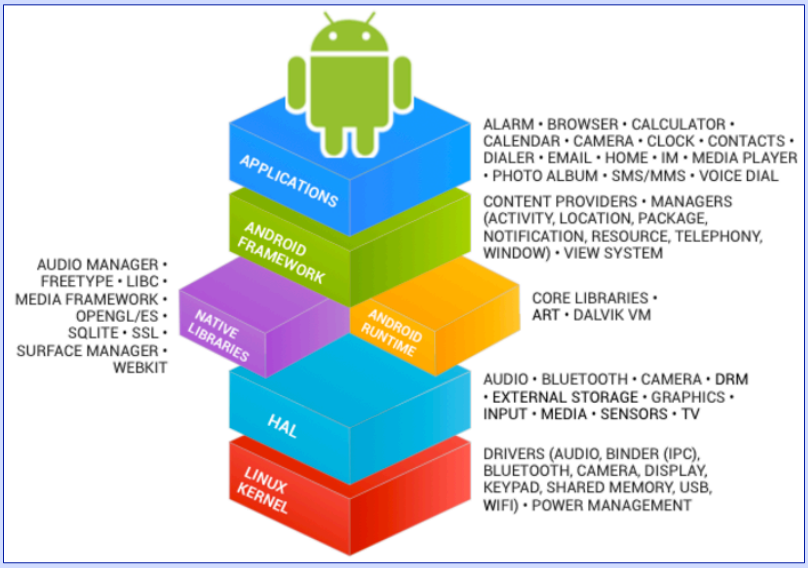
Android architecture

Layered structure: Each layer uses functionality from lower layers
(Source: http://developer.android.com/images/system-architecture.jpg)



Android stack: Another view

(Source: https://source.android.com/source/)



Linux kernel

- Lowest level
- Not quite a standard Linux
- Linux portion: Standard OS services including processor management, most memory management, file system, network I/O, permission structure, device drivers for radio, camera, etc.
- Android specific portion: Binder mechanism for Inter-Process Communication (IPC), Android-specific power management, Android shared memory, out-of memory behavior

Native libraries

- Next level up from Linux kernel
- Written in C/C++
- Core of performance-sensitive functions
  - Bionic (standard C library based on BSD libc)
  - Players of A/V media (aka the Media library)
  - SQLite database
  - WebKit (browser layout engine)
  - Display management, etc.

Also in native library layer

- Dalvik
  - Replaces Java Virtual Machine in Android, less computationally expensive than traditional JVM
  - Chief designer: Dan Bornstein (Google)
  - Executes bytecodes contained in .dex or .odex files
  - Two stage compilation (1st stage: standard compilation into Java bytcodes, 2nd stage: **dx** tool generates dex from JVM bytecodes)
  - Replaced by Android Run-Time (ART) in Lollypop
  - Faster execution, smaller image size than standard JVM
  - Additional info in Dan Bornstein's excellent video (1 hour): https://sites.google.com/site/io/dalvik-vm-internals

Also in native library layer (cont'd)

- Java libraries
  - Java Class Library: java.* and javax.* classes
  - Utilities for managing app life cycle — android.* classes
  - Internet/Web services — org.*
  - Unit testing — junit.*

Application framework layer (Java)

- Frameworks that you will need to write your apps
  - Define app components as extensions (e.g., Java subclasses) of basic functionality supported by classes in this layer
- Examples of frameworks
  - Activity manager — Manage all kind of components in running apps, including each activity stack
  - Package manager — DB of all apps currently installed on device
  - Window manager — Manage windows and subwindows appearing on display
  - View System — Common GUI elements (text fields, buttons, lists, etc.)

More frameworks

- Examples of frameworks (cont'd)
  - Resource manager — Manage layout files, strings, graphics, pictures in each app

    Resources (e.g., strings) kept separate from source code to support switching among multiple configurations
  - Telephony manager — Access to telephony services (monitor status, register for incoming calls, etc.)
  - ContentProvider — Share data between different apps, (e.g., contacts)
  - NotificationManager — Allow apps to place information in the notification bar and the pull-down "drawer"
  - etc.

App layer

- The apps that run on your phone
- Your apps will go in this layer as well
- Popular existing apps: Phone, IM (MMS), Browsers, Contacts, Calendar, Email, Media player, Camera, Albums, Alarm clock, etc.
- Tightly integrated with lower layer, but not wired in OS
- You are allowed to define your own apps for the same functionality as the predefined apps, override predefined apps
- App code == A number of callback classes and methods
  - App execution weaves in and out of your code and OS code
  - No *main()* method

App components

- An app consists of a set of *components*
- There are four kinds of components in Android
  1. **Activities** – Manage user interactions on device's display
  2. **Broadcast receivers** – Global "listeners"
  3. **Content providers** – Sharing persistent data among apps
  4. **Services** – Long running actions and background actions
- Most apps have at least one activity, but apps don't have to have one
- Implementation note: Each component is defined as an abstract Java class in framework layer; you define your app component by subclassing the abstract class from the framework

```
public class MyReceiver extends BroadcastReceiver { ...
```


App layer (cont'd)

- Each app runs in its own Unix process for security reasons
- Each app is associated with its own package
- Apps can communicate in various ways
  - *Intent*: A system-wide message describing an action to be performed
  - Example: Email app uses intent to launch browser after user selection

What are activities?

- Activity = *Single, focused thing that the user can do*
- Each activity has a window, traditionally occupying entire device display (no longer true as of Nougat)
- Typically, multiple activities in an app

  Example: email app will have separate activities for (1) displaying list of messages, (2) composing a message, (3) reading a message, etc.
- Example: class *AllInOneActivity* defines welcome screen for *Calendar* app
- Sources:

  http://developer.android.com/reference/android/app/Activity.html
  http://developer.android.com/guide/components/activities.html

What are services?

- Tasks that run **in the background** (without interacting with device user)
- Main goals:
  1. Support long-running operations (e.g., playing music)

     User can do other things while music is playing
  2. Interact with remote processes (e.g., synchronizing email or contacts with a remote server)
- Example of a service: *MediaPlaybackService* class
- Sources:

  http://developer.android.com/reference/android/app/Service.html
  http://developer.android.com/guide/components/services.html

On broadcast receivers

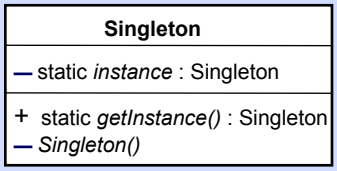
- Components that listen for special messages called *Intents* and respond to those messages
- Intents are broadcast by applications, e.g., using app framework method *sendBroadcast()* and related methods
- Receiver's *onReceive()* method specifies actions in response to intent
- Example: Messaging application
  - SMS arrives, OS broadcasts intent SMS_RECEIVED
  - Broadcast receiver in messaging app starts service that downloads and stores SMS content locally (e.g., see file *SmsReceiver.java*)
- Source: http://developer.android.com/reference/android/content/BroadcastReceiver.html

Singleton pattern

- Sometimes you want exactly one instance of a given class, e.g.,
  - Android wants at most one messaging app running on a device at any given time
- Goals:
  - Create the instance, e.g., when the system boots
  - Forbid clients of singleton class from creating more instances
- Singleton class must control object creation
- A Pattern tied to a programming language
  - Different languages do singletons differently
  - *Idiom* = A low-level pattern = A programming cliché

Class diagram of *Singleton*

- Singleton class uses **private, static** field to hold unique instance
- Singleton constructors declared private, to prevent instance creation by clients
- Class defines public, **static** method *getInstance(),* which either returns existing instance or creates it if there isn't one
- Clients obtain instance with expression *Singleton.getInstance()*
- A Java question: *SmsReceiver* is a singleton, but with no constructors. What does this mean?



Singleton and *SmsReceiver* class

1. Compiler generates automatically a no-arg constructor for class *SmsReceiver*
2. Automatically-generated constructor invokes superclass's constructor
   - Superclass *BroadcastReceiver* does have a public, no-arg constructor
   - This means that new instances of *SmsReceiver* can in fact be created at run-time (e.g., using syntax: `new SmsReceiver()`)
   - This is possible according to code in *SmsReceiver.java*
3. However, compiler-generated no-arg constructor has *package* access level by default
4. Consequence: Only *Mms* app can create instance of *SmsReceiver* class

On content providers

- Components that encapsulate data sets (e.g., databases, files, web repositories, etc.)
  - Provide API similar to a relational database, with support for SQL operations (e.g., select, update, project, join, etc.)
  - Use when persistent data shared by multiple apps
  - Can handle low-level IPC (Inter-Process Communication)
- Example: Browser app uses class *BrowserProvider* to store bookmarks and recently searched strings
- Sources:

  http://developer.android.com/reference/android/content/ContentProvider.html

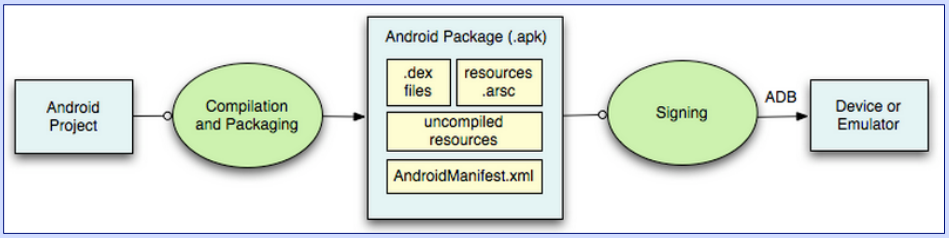
  http://developer.android.com/guide/topics/providers/content-providers.html

Developing Android apps

A five step process:

1. Define app resources (e.g., strings, colors, pictures, video segments, etc.)
2. Write Java code for app classes, incorporate resources into code
3. Compile and package your app
   - This will produce an .apk (application package) file containing both your .dex files (Dalvik executables), packaged resources, and AndroidManifest files
4. Sign the app (debug vs. release mode)
5. Install and run the app (either on actual device or emulator)

Flow of application building process



- Sources:

  http://developer.android.com/tools/building/index.html

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

Useful links

- Porter's Coursera slides
  - https://www.coursera.org/learn/android-programming/lecture/tebCM/introduction-to-the-android-platform (cut and paste in URL window)
- Source code repository
  - https://android.googlesource.com/
  - https://source.android.com/source/
  - E.g., class *SmsReceiver* for Lollipop V5.1.1 is at URL:

    https://android.googlesource.com/platform/packages/apps/Mms/+/android-5.1.1_r33/src/com/android/mms/transaction/SmsReceiver.java

Course outline

1. Activities, GUI elements and intents
2. Fragments
3. Broadcast receivers and broadcast intents
4. Android multithreading
5. Background services
6. Files, SQLite databases and content providers
7. Networking and JSON
8. Sensors, location and maps