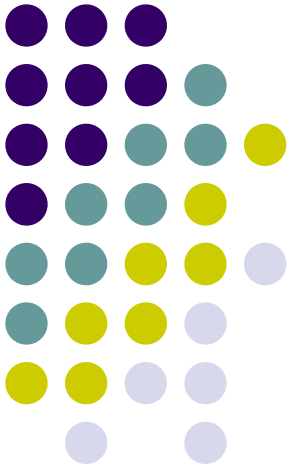
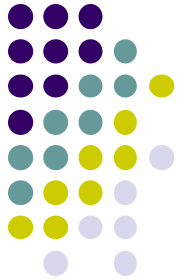
Mobile Programming

Introduction to Android



What is Android?



- Android is a software stack for mobile devices that includes an operating system, middleware and key applications.
- The Android SDK provides the tools and APIs necessary to begin developing applications on the Android platform using the Java programming language.
- Android also began supporting Kotlin in 2017 as an alternative dev. Language.

Brief History

- Google bought Android Inc. from Andy Rubin (Palo Alto, fnd. 2003) in 2005 (50 mil. $)
- In 2007 Open Handset Alliance (OHA) was formed by Google, HTC, Sony, LG, Samsung, T-Mobile and more
- Android was unveiled as OHA's first product, a mobile device platform built on the Linux kernel version 2.6.25 (HTC Dream is the first product)
- For version history check:
  - http://www.android.com/history/

Andy Rubin, illustration by Mike Nudelman/Business Insider

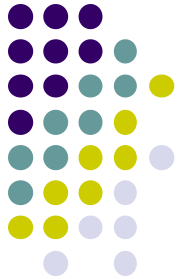
Early Devices 2005-2008

HTC Dream or T-Mobile G1

Motorolla Droid

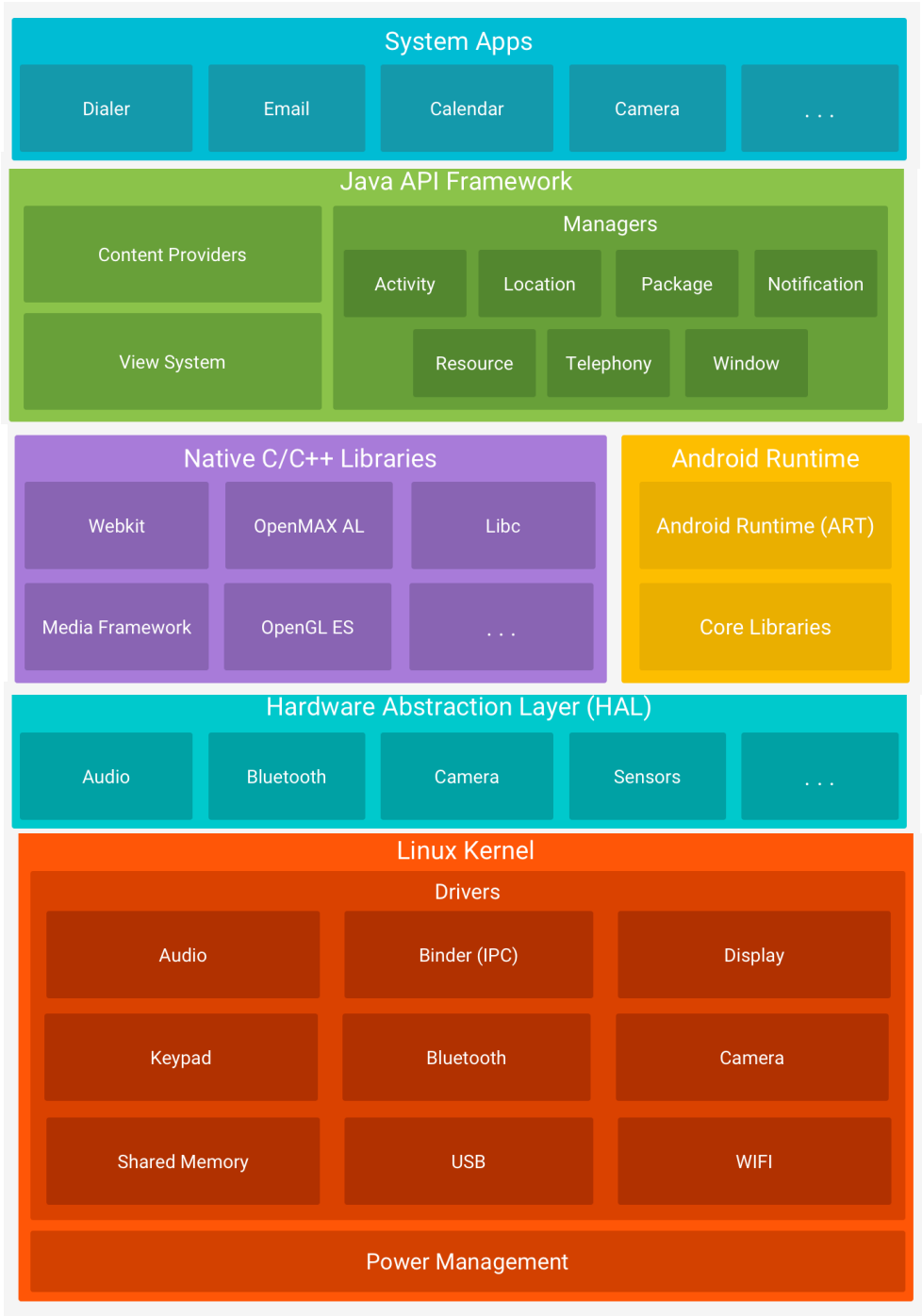
Iphone 1

Features

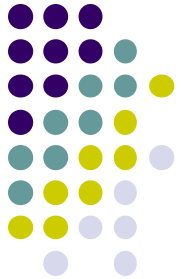


- **Application framework** enabling reuse and replacement of components
- **Dalvik virtual machine / ART** optimized for mobile devices
- **Integrated browser** based on the open source WebKit engine
- **Optimized graphics** powered by a custom 2D graphics library; 3D graphics based on the OpenGL ES 1.0, 1,1, 2.0 specification (hardware acceleration optional)
- **SQLite** for structured data storage
- **Media support** for common audio, video, and still image formats (MPEG4, H.264, MP3, AAC, AMR, JPG, PNG, GIF)
- **GSM Telephony** (hardware dependent)
- **Bluetooth, EDGE, 3-4-5G, and WiFi** (hardware dependent)
- **Camera, GPS, compass, and accelerometer** (hardware dependent)
- **Rich development environment** including a device emulator, tools for debugging, memory and performance profiling, and a plugin for the Android Studio (An IntelliJ Idea Version)

Android Architecture

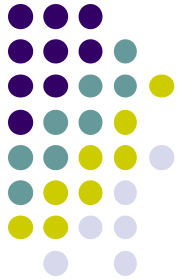


Application Fundamentals



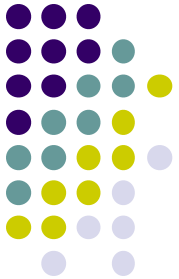
- Written in Java / Kotlin
- The Android SDK tools compile the code into Android Package (.apk file)
- The Android operating system is a multi-user Linux system in which each application is a different user.
- Each process has its own virtual machine (VM) , isolated from other applications

Android Runtime



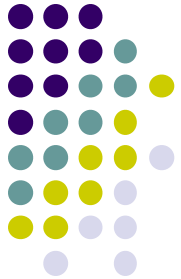
- Android includes a set of core libraries that provides most of the functionality available in the core libraries of the Java programming language
- Every Android application runs in its own process, with its own instance of the Dalvik virtual machine or ART. (**Sandboxing**)
- **The Dalvik/ART VM executes files in the Dalvik Executable (.dex)**
- The Dalvik/ART VM relies on the Linux kernel for underlying functionality such as threading and low-level memory management.

Why Dalvik or ART?



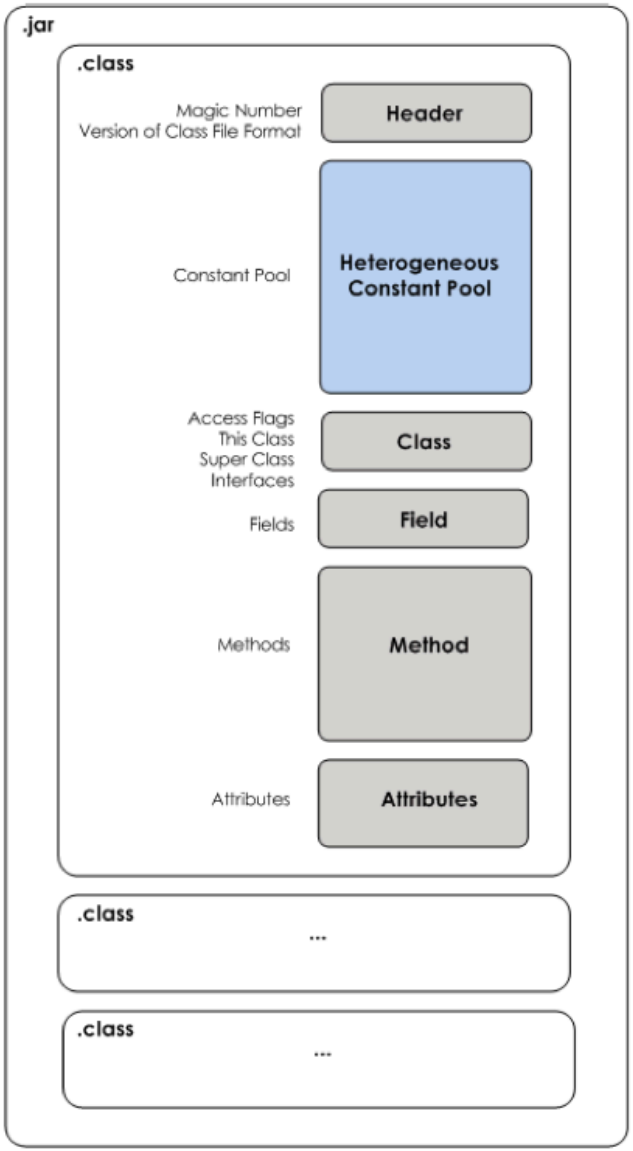
- Limited processor speed
- Limited RAM
- No swap space
- Battery Powered
- Diverse set of devices
- Sandboxed environment

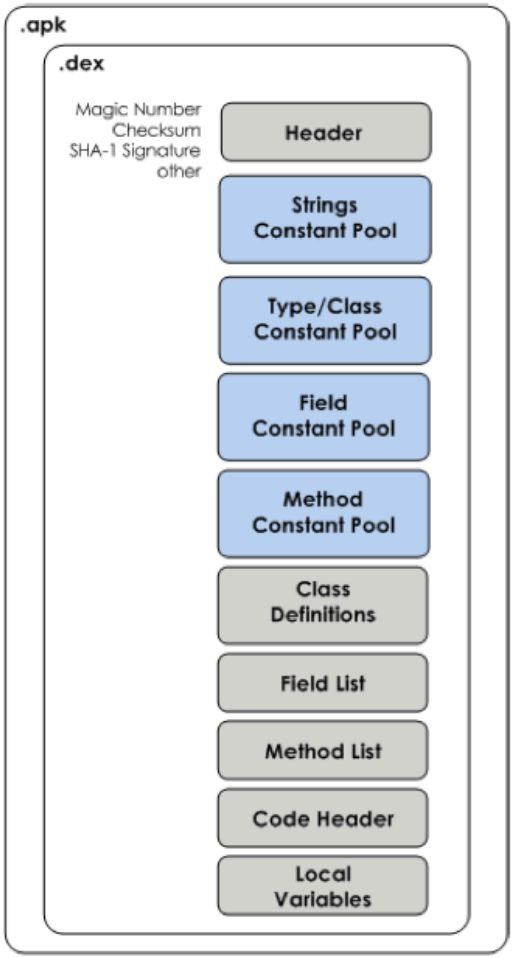
DEX File Format



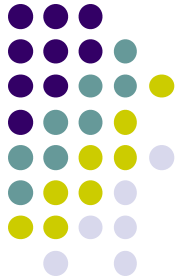
- Standard JVM → a .class file for each class (static inner, anonymous, etc)
- DEX → .class converted to .dex by Dx tool, all classes in one file
- The .dex file has been optimized for memory usage and the design is primarily driven by sharing of data.

.class vs .dex



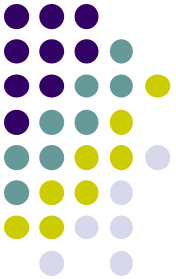


A New VM: ART



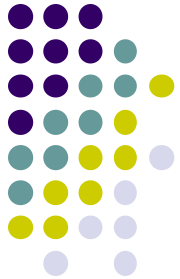
- With KitKat a new VM called Android Runtime is released, default VM after Lollipop
- ART has several main features compared to Dalvik:
  - Ahead-of-Time (AOT) compilation, which improves speed (particularly startup time) and reduces memory footprint (no **JIT**)
  - Improved Garbage Collection (GC)
  - Improved memory usage
- Not all devices support ART yet, so Dalvik may still be in use

AOT vs JIT Compilation



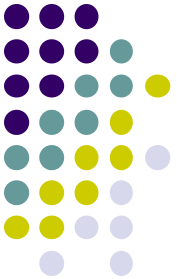
- They refer to when the compilation takes place.
- **AOT** compiles at install time, get use of .oat files (Increases app performance)
- **JIT** compiles at runtime, get use of .dex files according to component needs (Decreases app performance)
- **ART** uses both in an optimized way.

The Zygote



- The application runner OR Virtual machine manager
- Controls applications lifecycle
- Enables both sharing of code across VM instances and to provide fast startup time of new VM instances
- Starts with system boot

Security



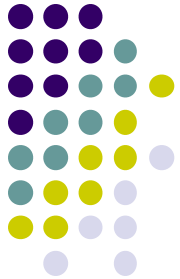
- Sandboxed environment!

> *No application, by default, has permission to perform any operations that would adversely impact other applications, the operating system, or the user. This includes reading or writing the user's private data (such as contacts or e-mails), reading or writing another application's files, performing network access, keeping the device awake, etc.*

How to Develop

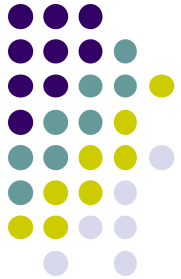
- Download the Android Studio (IntelliJ IDEA) using the link below
  - https://developer.android.com/studio/install.html
  - Download / Update available packages in the SDK
- Create an emulator device
  - From SDK, Virtual Devices

Application Components



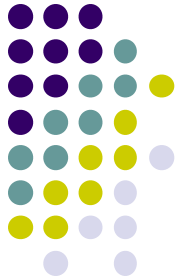
- **Activities**
  - An *Activity* represents a single screen with a user interface. (subclass of Activity)
- **Services**
  - A *service* is a component that runs in the background to perform long-running operations or to perform work for remote processes (ex. Fetch data from network, get location, subclass of Service)

Application Components (cont'd)



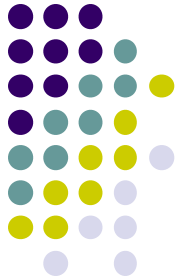
- **Content Providers**
  - A *content provider* manages a shared set of application data. You can store the data in the file system, an SQLite database, on the web, or any other persistent storage location your application can access. (Subclass of ContentProvider)
- **Broadcast Receivers**
  - A *broadcast receiver* is a component that responds to system-wide broadcast announcements. (ex. Screen turned off, battery low. Subclas of BroadcastReceiver)

Activating Components



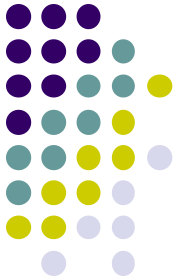
- Activities, services and broadcast receivers are activated by an asynchronous message called ***Intent***
- For activities and services, an intent defines the action to perform (for example, to "view" or "send" something)
- For broadcast receivers, the intent simply defines the announcement being broadcast (for example, a broadcast to indicate the device battery is low includes only a known action string that indicates "battery is low")
- Content providers are activated by ContentResolver

Activating Components (cont'd)



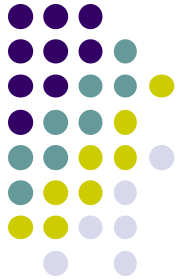
- You can start an **activity** (or give it something new to do) by passing an Intent to startActivity() or startActivityForResult() (when you want the activity to return a result).
- You can start a **service** (or give new instructions to an ongoing service) by passing an Intent to startService(). Or you can bind to the service by passing an Intent to bindService().
- You can initiate a **broadcast** by passing an Intent to methods like sendBroadcast(), sendOrderedBroadcast(), or sendStickyBroadcast().
- You can perform a **query** to a content provider by calling query() on a ContentResolver.

Android Project



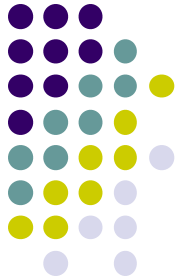
- manifests: Contain AndroidManifest.xml
- java : Java source files
- res: All static resources
  - drawable: images and background XMLs
  - mipmap: App icons
  - layout: Layout XML files.
  - values: color, String, style resources
- Gradle Scripts: Contains Gradle wrapper files

.apk File Contents



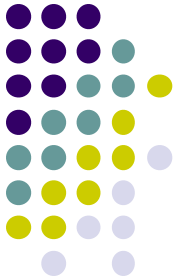
- APK
  - META-INF
  - lib
  - res
  - assets
  - AndroidManifest.xml
  - classes.dex
  - resources.arsc

Application Resources



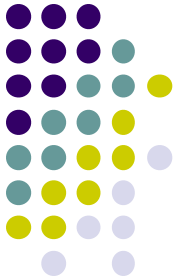
- Animations, menus, styles, colors, and the layout of activity user interfaces are defined with XML resource files
- Each resource has a unique id
- Example:
  - Jpg/png images → res/drawable
  - XML layouts → res/layout and res/layout-land
  - Strings → res/values/strings.xml
  - XML Data → res/xml
  - Raw data → res/raw
- Class R contains all the resources automatically and can be accessed statically from code

The Manifest File



- All components must be declared in the AndroidManifest.xml
- Resides at the root of the app directory
  - Identify any user permissions the application requires, such as Internet access or read-access to the user's contacts.
  - Declare the minimum API Level required by the application, based on which APIs the application uses.
  - Declare hardware and software features used or required by the application, such as a camera, bluetooth services, or a multitouch screen.
  - API libraries the application needs to be linked against (other than the Android framework APIs), such as the Google Maps library.
  - And more

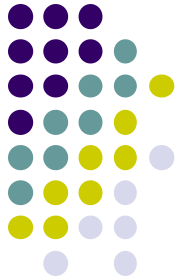
Declaring Components in the Manifest



```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
    <application android:icon="@drawable/app_icon.png" ... >
        <activity android:name="com.example.project.ExampleActivity"
                  android:label="@string/example_label" ... >
        </activity>
        ...
    </application>
</manifest>
```

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

Aiming at a Target



- Android apps are developed for a target level (ex: target → level 28 means Android 9 Pie)
  - Ask for only what you need, choose the minimum API level meeting your needs
  - Test on as many targets as you can and that are possible (create on level 11 test on 12, 13, 14, ..)
  - Check out the new target levels with each Android release (users are updating all the time)
  - There is no substitute for testing on hardware (hardware and emulator are not the same)

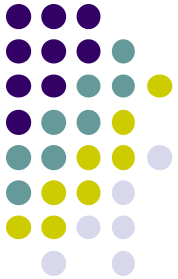
Platform Versions

ANDROID PLATFORM VERSION	API LEVEL	CUMULATIVE DISTRIBUTION
4.1 Jelly Bean	16	
4.2 Jelly Bean	17	99.9%
4.3 Jelly Bean	18	99.7%
4.4 KitKat	19	99.7%
5.0 Lollipop	21	98.8%
5.1 Lollipop	22	98.4%
6.0 Marshmallow	23	96.2%
7.0 Nougat	24	92.7%
7.1 Nougat	25	90.4%
8.0 Oreo	26	88.2%
8.1 Oreo	27	85.2%
9.0 Pie	28	77.3%
10. Q	29	62.8%
11. R	30	40.5%
12. S	31	13.5%

Android APIs are forward compatible; so least is the version number, most is the coverage!

http://developer.android.com/resources/dashboard/platform-versions.html

Compiler Preferences

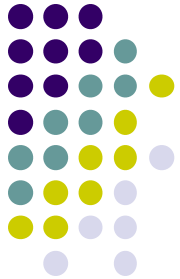


- set in build.gradle (module)
- **compileSdkVersion**: Used to inform developer at compile time, use the latest version
- **minSdkVersion**: Lower bound of the app, used in PlayStore
- **targetSdkVersion**: Main way Android provides forward compatibility, used to check deprecations, new features, etc

Ideal Settings

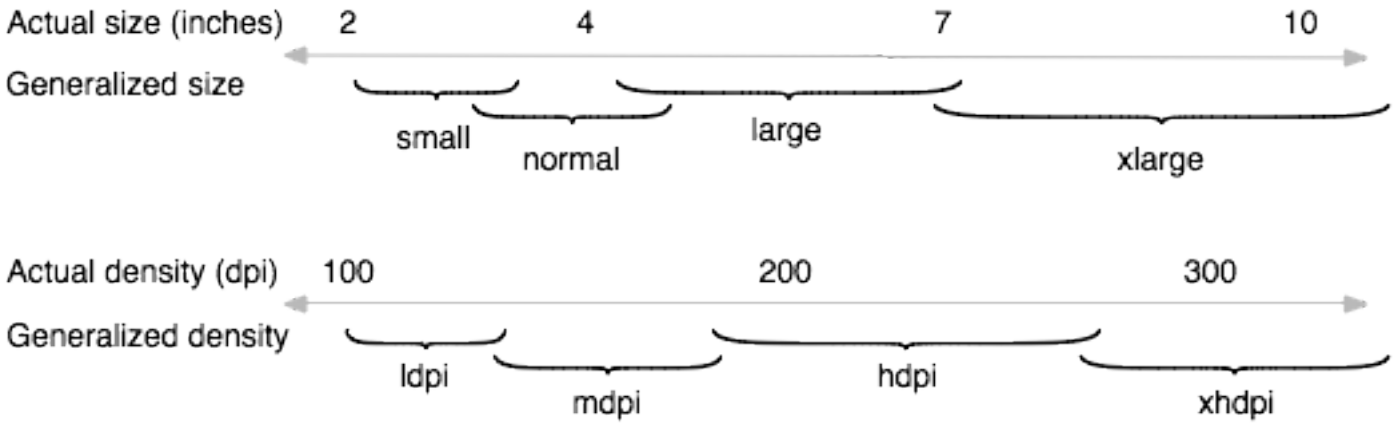
```
minSdkVersion (lowest possible) <=
    targetSdkVersion == compileSdkVersion (latest SDK)
```

Screen Size and Density



- *Screen Size:* Actual physical size, measured as the screen's diagonal in inches.
  - small, normal, large and extra large
- *Screen Density:* The quantity of pixels within a physical area of the screen; usually referred to as dpi (dots per inch)
  - low, medium, high, extra-high, extra-extra-high, extra-extra-extra-high
- *Density-independent pixel (dp)* : A virtual pixel unit that you should use when defining UI layout, to express layout dimensions or position in a density-independent way.

Display Strategy

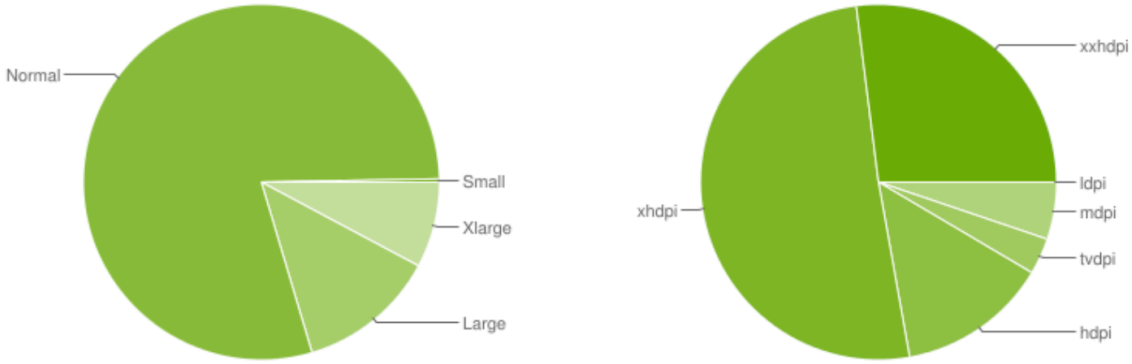


- *xlarge* screens are at least 960dp x 720dp
- *large* screens are at least 640dp x 480dp
- *normal* screens are at least 470dp x 320dp
- *small* screens are at least 426dp x 320dp

- Sizes are abstracted to applications
- Pick the smallest size and test with larger
- For more: http://developer.android.com/guide/practices/screens_support.html

Screen Resolutions / Sizes

	ldpi	mdpi	tvdpi	hdpi	xhdpi	xxhdpi	Total
Small					0.3%		0.3%
Normal		0.1%	0.2%	9.4%	44.6%	25.0%	79.3%
Large		1.2%	3.0%	0.8%	5.6%	2.0%	12.6%
Xlarge		3.9%	0.1%	3.5%	0.3%		7.8%
Total	0.0%	5.2%	3.3%	13.7%	50.8%	27.0%	



Data collected during a 7-day period ending on June 24, 2022.
Any screen configurations with less than 0.1% distribution are not shown.

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

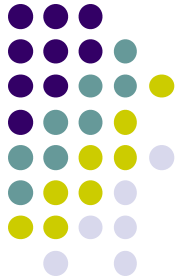
Naming Resources for Sizes

```
res/layout/my_layout.xml                // layout for normal screen size ("default")
res/layout-large/my_layout.xml          // layout for large screen size
res/layout-xlarge/my_layout.xml         // layout for extra-large screen size
res/layout-xlarge-land/my_layout.xml    // layout for extra-large in landscape orientation

res/drawable-mdpi/graphic.png           // bitmap for medium-density
res/drawable-hdpi/graphic.png           // bitmap for high-density
res/drawable-xhdpi/graphic.png          // bitmap for extra-high-density
res/drawable-xxhdpi/graphic.png         // bitmap for extra-extra-high-density

res/mipmap-mdpi/my_icon.png          // launcher icon for medium-density
res/mipmap-hdpi/my_icon.png          // launcher icon for high-density
res/mipmap-xhdpi/my_icon.png         // launcher icon for extra-high-density
res/mipmap-xxhdpi/my_icon.png        // launcher icon for extra-extra-high-density
res/mipmap-xxxhdpi/my_icon.png       // launcher icon for extra-extra-extra-high-density
```

Naming Resources for Sizes

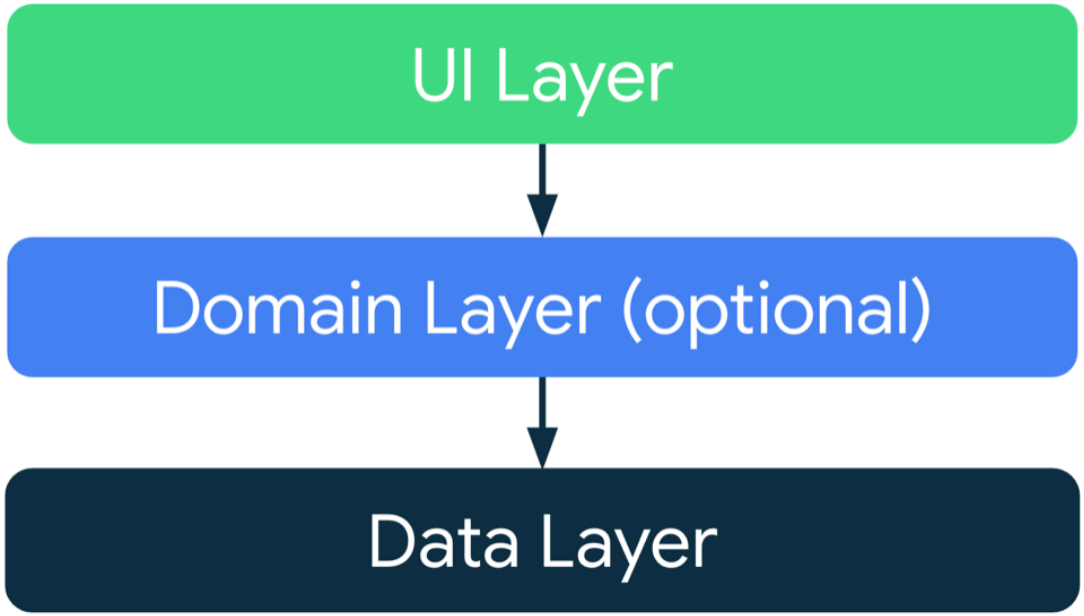


```
res/layout/main_activity.xml
# For handsets (smaller than 600dp available width)

res/layout-sw600dp/main_activity.xml
# For 7” tablets (600dp wide and bigger)
```

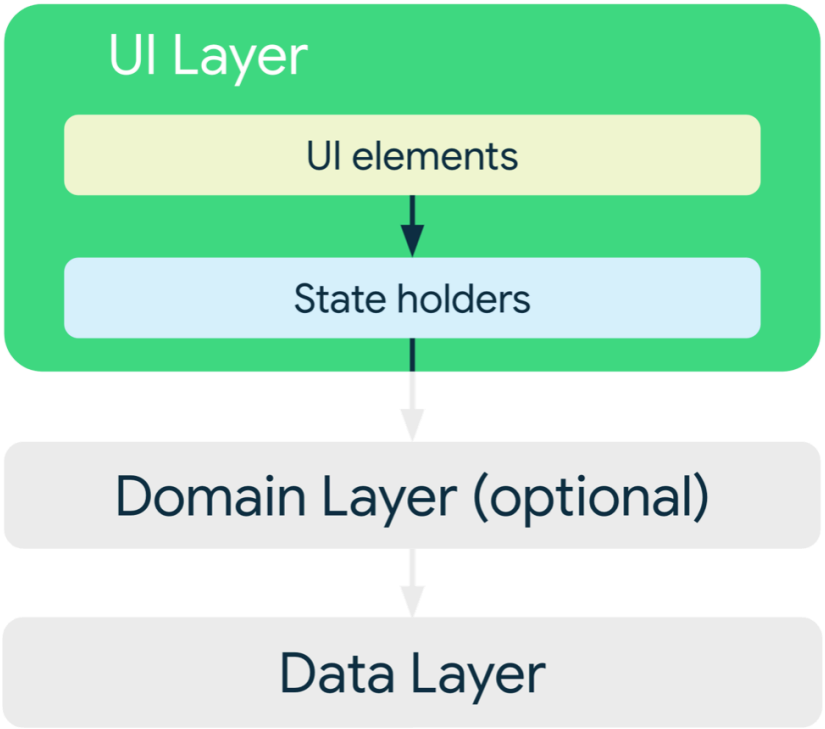
```
res/layout/main_activity.xml                # For handsets
res/layout-land/main_activity.xml           # For handsets in landscape
res/layout-sw600dp/main_activity.xml        # For 7” tablets
res/layout-sw600dp-land/main_activity.xml   # For 7” tablets in landscape
```

Recommended App Architecture



- The *UI layer* that displays application data on the screen.
- The *data layer* that contains the business logic of your app and exposes application data.
- You can add an additional layer called the domain layer to simplify and reuse the interactions between the UI and data layers.

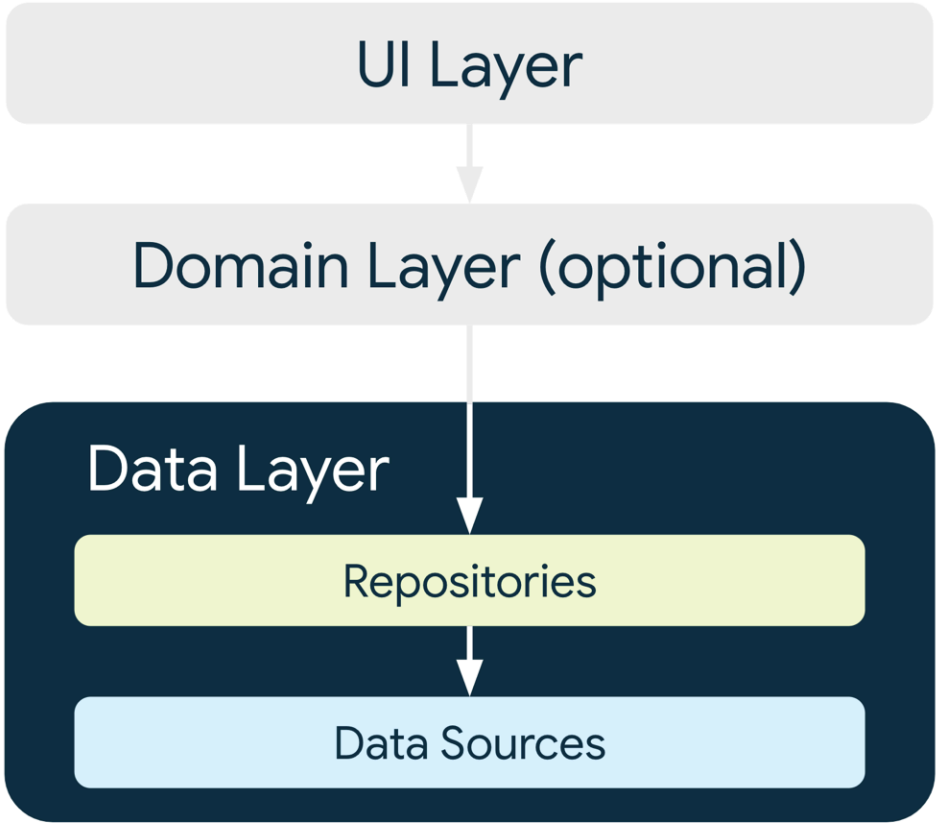
UI Layer



The UI layer is made up of two things:

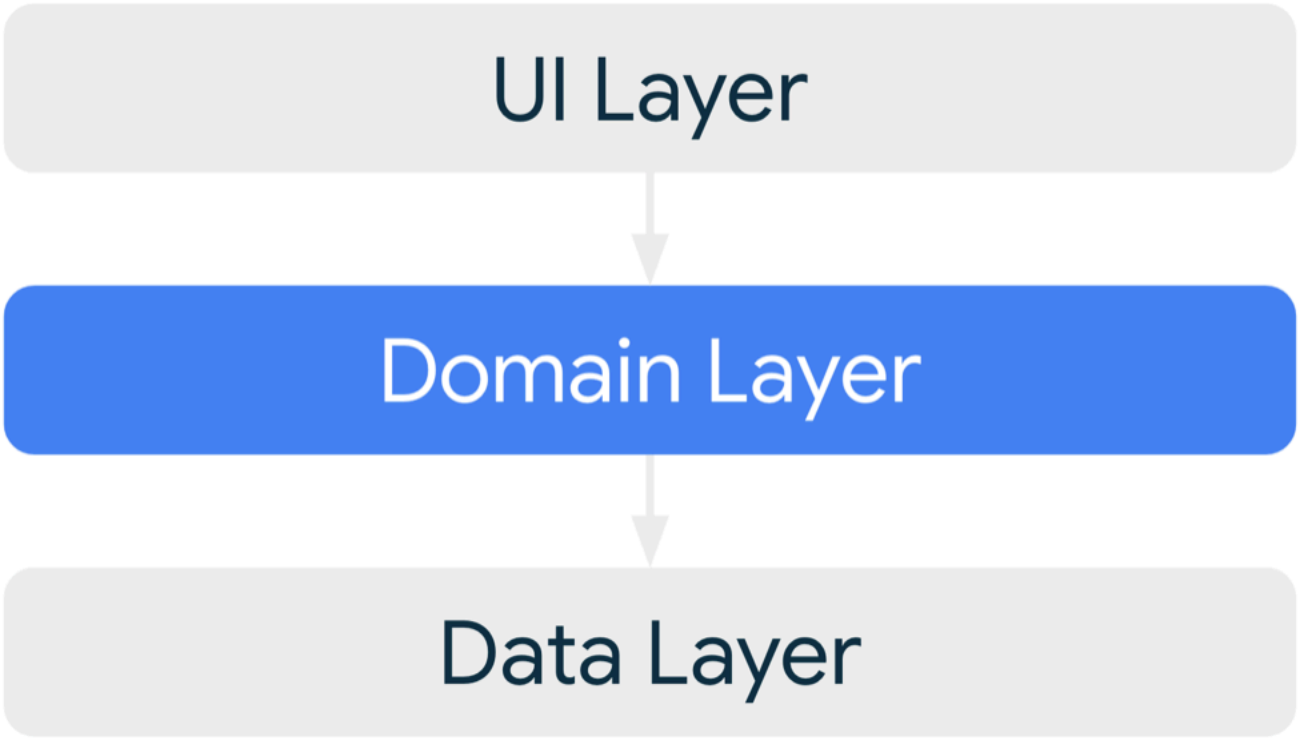
- UI elements that render the data on the screen. You build these elements using Views or Jetpack Compose functions.
- State holders (such as ViewModel classes) that hold data, expose it to the UI, and handle logic.

Data Layer



The data layer of an app contains the *business logic*. The business logic is what gives value to your app—it's made of rules that determine how your app creates, stores, and changes data.

Optional Domain Layer



The domain layer is responsible for encapsulating complex business logic, or simple business logic that is reused by multiple ViewModels. This layer is optional because not all apps will have these requirements. You should use it only when needed—for example, to handle complexity or favor reusability.