

Welcome to Mobile Software Development...

Lecture begins at 9 a.m.

# Mobile Software Development

**CMPU 3036 DT211C/3, DT282/3**

# Contact Details

- [david.leonard@tudublin.ie](mailto:david.leonard@tudublin.ie)

# Contact Details

- [david.leonard@tudublin.ie](mailto:david.leonard@tudublin.ie)
- Please be patient in awaiting a response, as there can be many emails requiring attention from different classes and staff members.



Inbox ( $\infty$ )



# Contact Details



- **Office Hours:** Fridays 2 p.m. – 5 p.m. (by **appointment** only!)

# Class Times

- **LECTURE Online** Friday 9 – 11 a.m.
- **LABS Online** Wednesday 9 – 11 a.m. (Me + TAs Giancarlo and Trinh)

# Class Etiquette

- 1 Recommendation...

# Class Etiquette

- 1 Recommendation...



# Me

- **Teaching:**  
Machine Learning, Data Mining, Data Wrangling, Business Systems  
Intelligence, Mobile Software Development, Programming.
- **Research:**  
Data analytics, Machine Learning

# You

- **First Year** - Procedural Programming – C
  - **Second Year** - Object Oriented Programming – Java
- Databases??
- Web: Client side HTML/ CSS/...
- Server side programming...

# 1. Module Details

# Learning Outcomes

- Design and Build Mobile Apps



# Learning Outcomes

- Design and Build Mobile Apps – Using **ANDROID** Environment

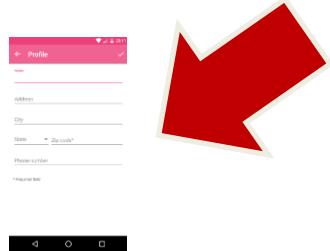


# Learning Outcomes

- Implement **USER INTERFACE** Components

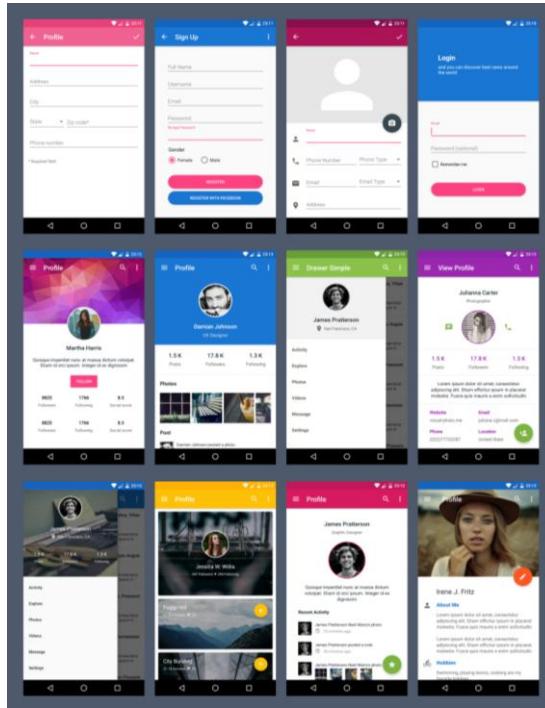
# Learning Outcomes

- Implement **USER INTERFACE** Components



# Learning Outcomes

- Implement **USER INTERFACE** Components



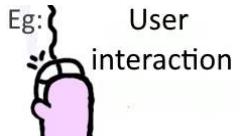
There are lots of  
different UI components  
in Android!

# Learning Outcomes

- Implement **EVENT DRIVEN** Programming

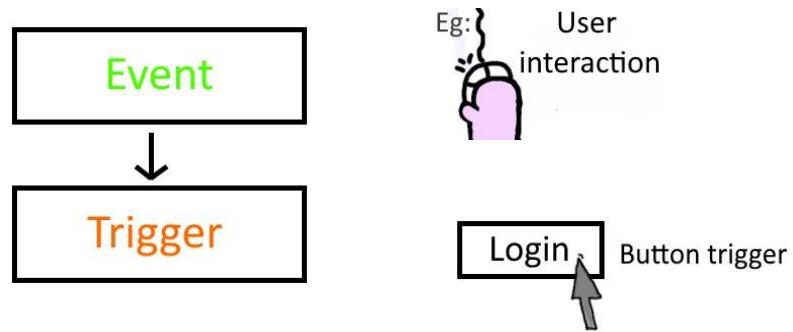
# Learning Outcomes

- Implement **EVENT DRIVEN** Programming



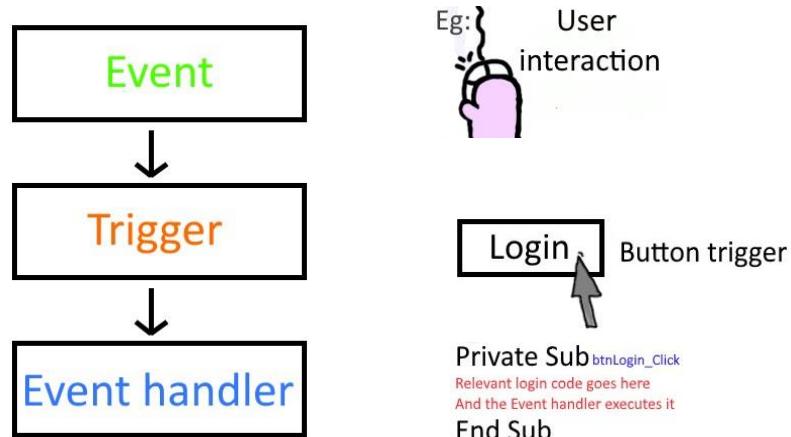
# Learning Outcomes

- Implement **EVENT DRIVEN** Programming



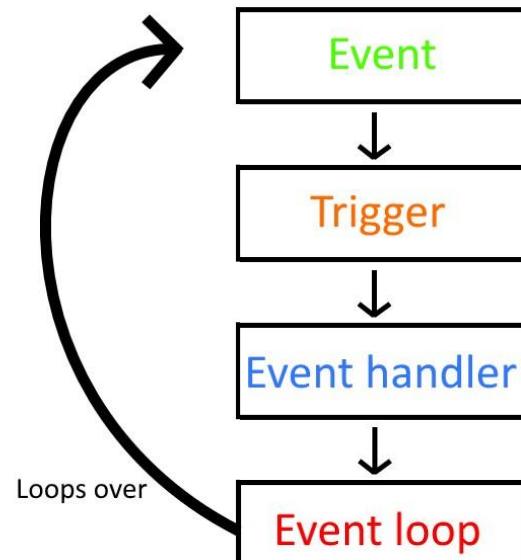
# Learning Outcomes

- Implement **EVENT DRIVEN** Programming



# Learning Outcomes

- Implement **EVENT DRIVEN** Programming

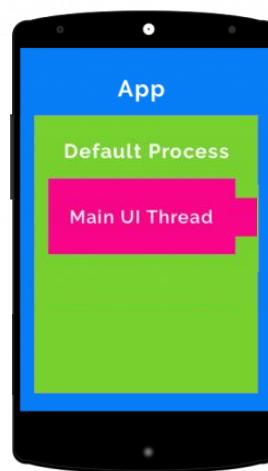


# Learning Outcomes

- Create **THREADS** / asynchronous processes

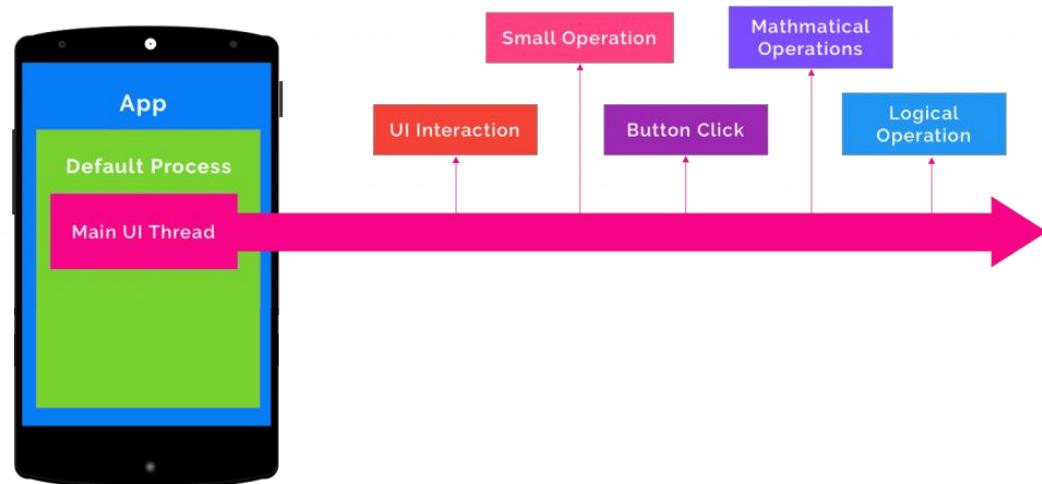
# Learning Outcomes

- Create **THREADS** / asynchronous processes



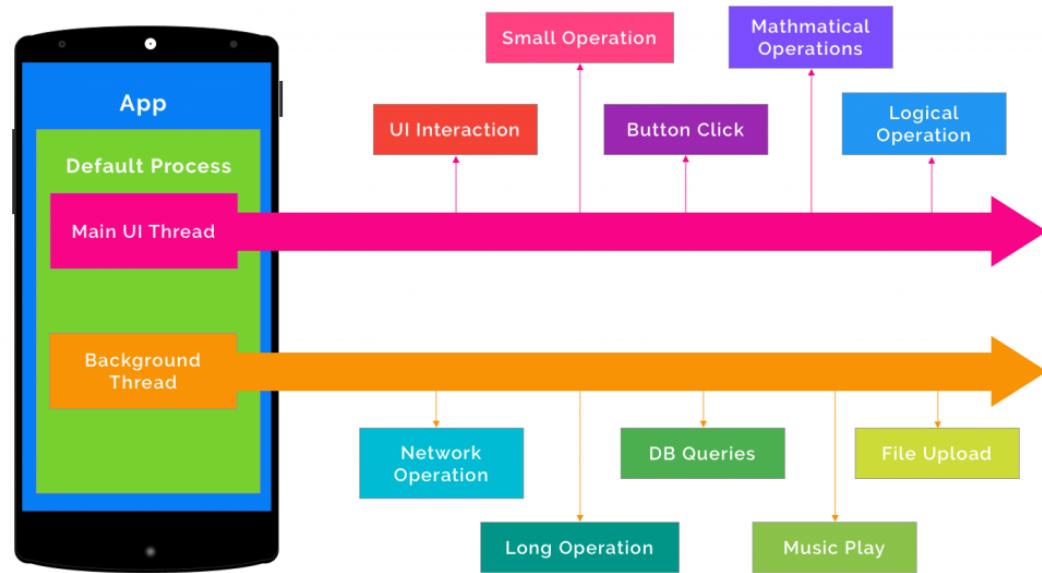
# Learning Outcomes

- Create **THREADS** / asynchronous processes



# Learning Outcomes

- Create **THREADS** / asynchronous processes



# Learning Outcomes

- Use data sources / **DATABASES.**

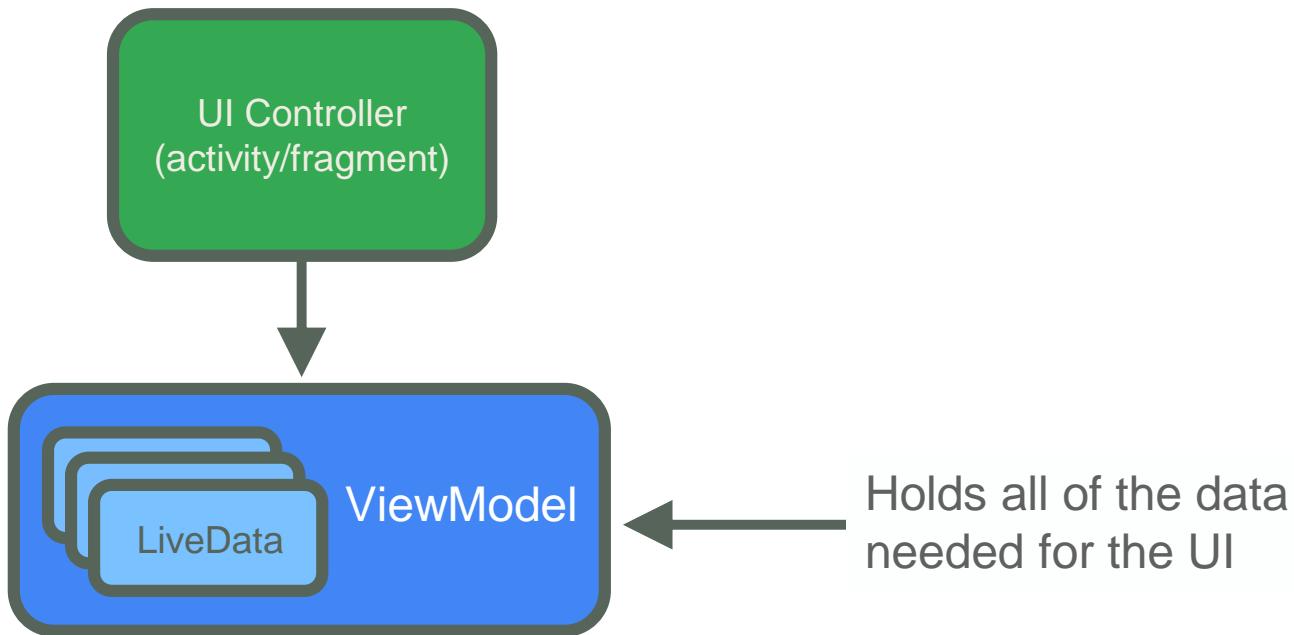
# Learning Outcomes

- Use data sources / **DATABASES.**



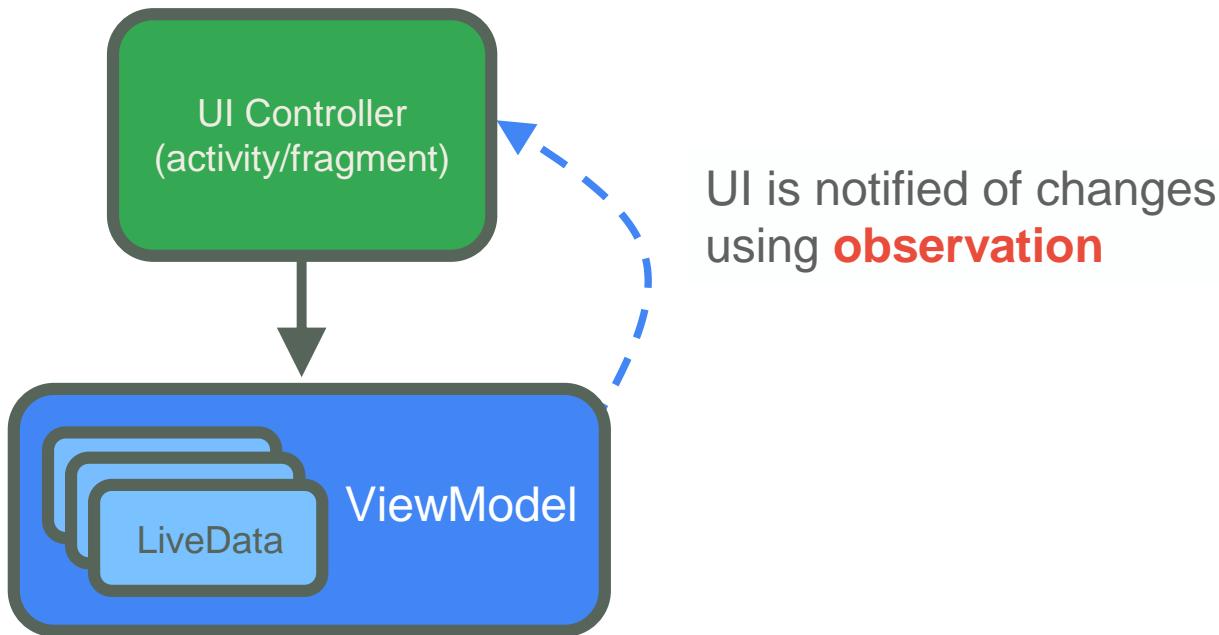
# Learning Outcomes

- Use data sources / **DATABASES.**



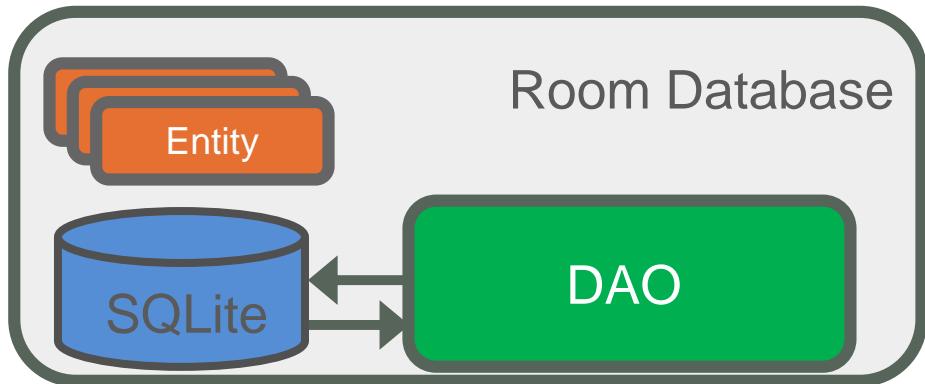
# Learning Outcomes

- Use data sources / **DATABASES.**



# Learning Outcomes

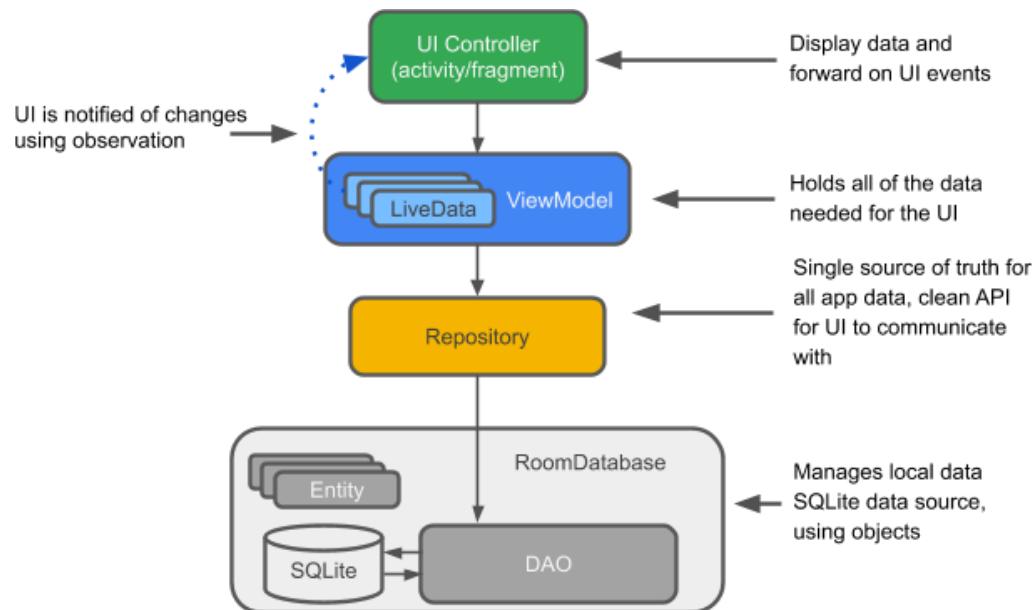
- Use data sources / **DATABASES**.



← Manages local data  
SQLite data source  
using **OBJECTS**

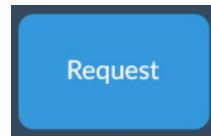
# Learning Outcomes

- Use data sources / **DATABASES.**



# Learning Outcomes

- Use remote **NETWORK** resources



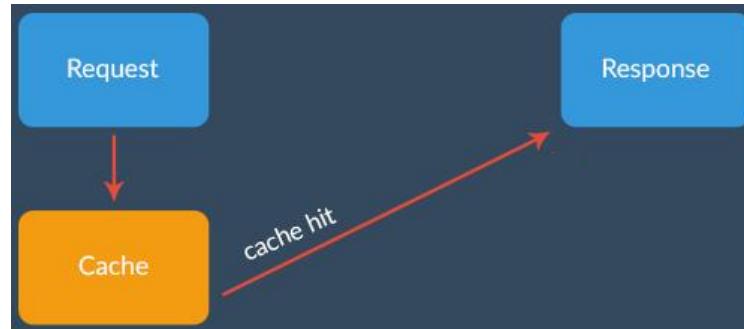
# Learning Outcomes

- Use remote **NETWORK** resources



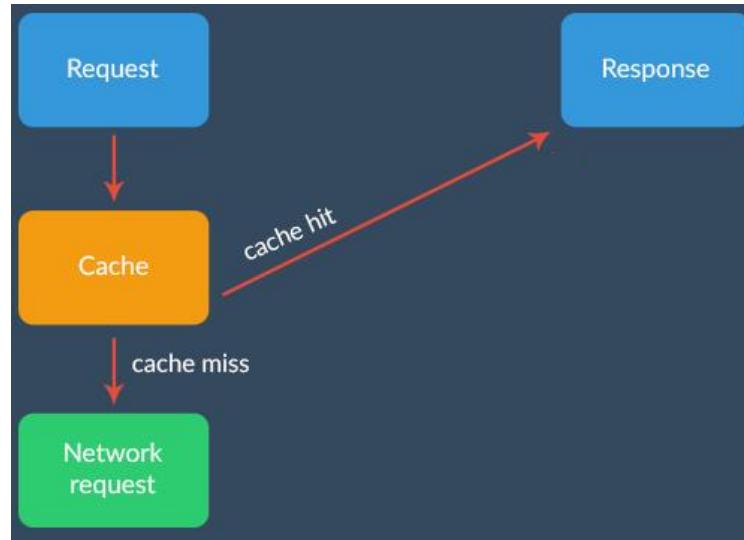
# Learning Outcomes

- Use remote **NETWORK** resources



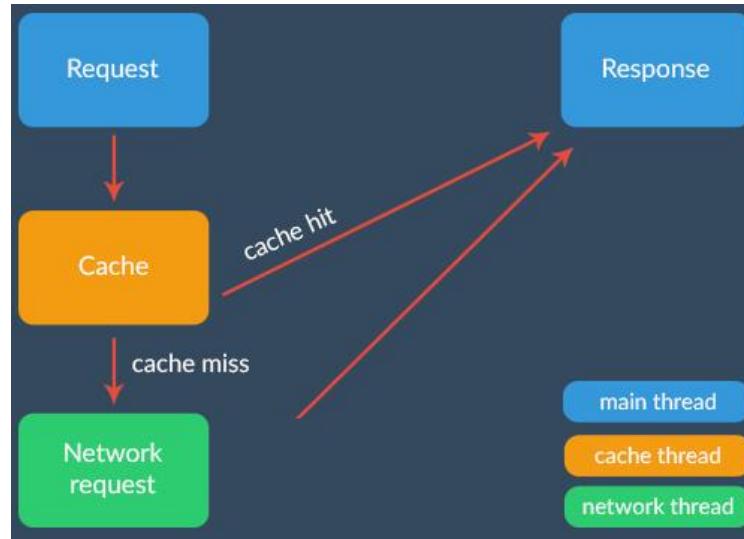
# Learning Outcomes

- Use remote **NETWORK** resources



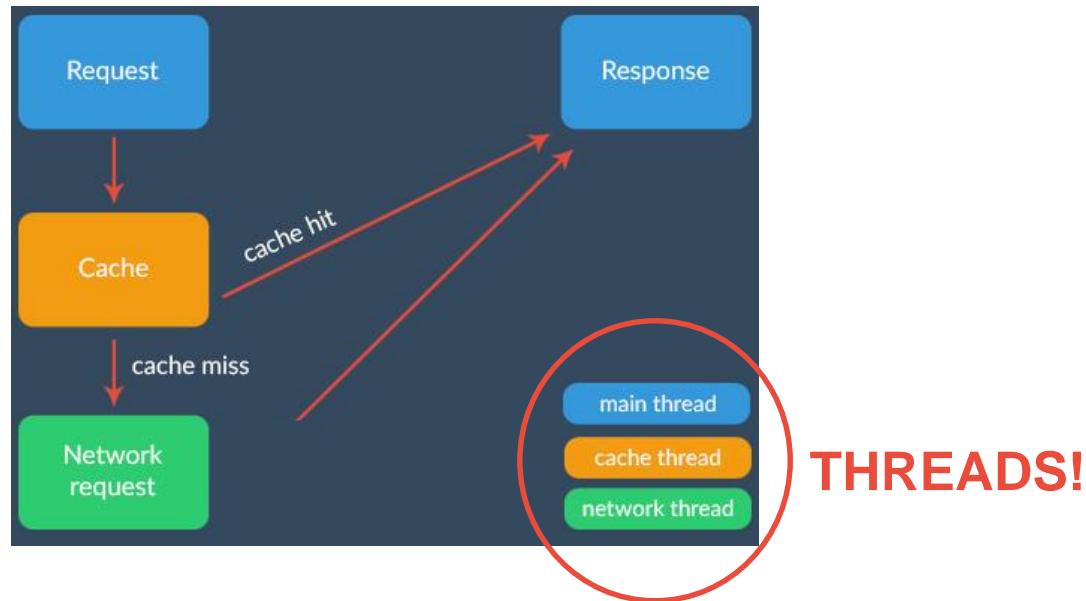
# Learning Outcomes

- Use remote **NETWORK** resources



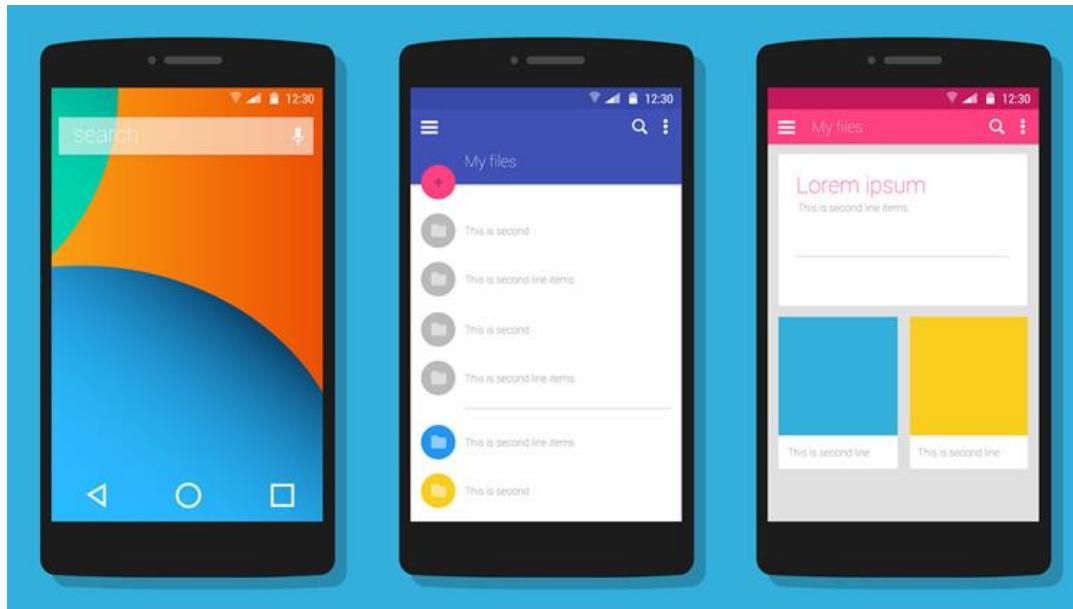
# Learning Outcomes

- Use remote **NETWORK** resources



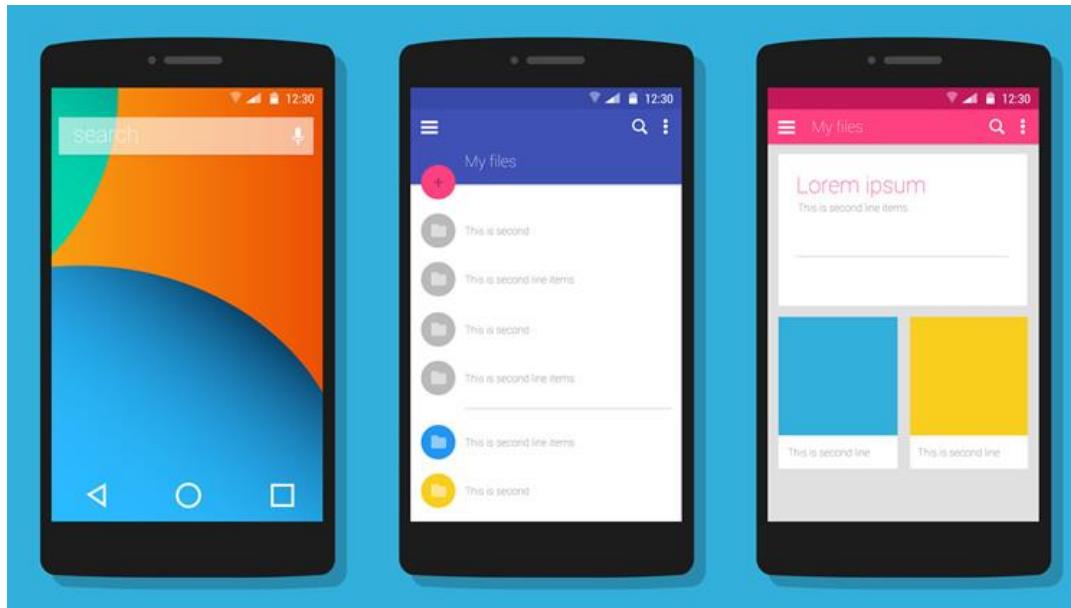
# Learning Outcomes

- Develop **USABLE** applications



# Learning Outcomes

- Develop **USABLE** applications



A user interface is like a joke. If you have to explain it, it's not that good.

# Topics

- Java / XML layouts
- Containers
- Event programming
- UI components
- Lists / adapters
- Using databases in your app
- Using networked resources
- Threaded programming
- Location based apps and Maps

# Resources

- Android **DEVELOPERS** Website  
<https://developer.android.com/>



Platform

Android Studio

More ▾

## Build anything on Android



Search

# Latest Android News

## FEATURED

### Android 11 is here!

Android 11 is now officially available to the Android Open Source Project (AOSP) and on its way to users! Get your apps ready for this new version of Android!

android 11

[LEARN MORE](#)

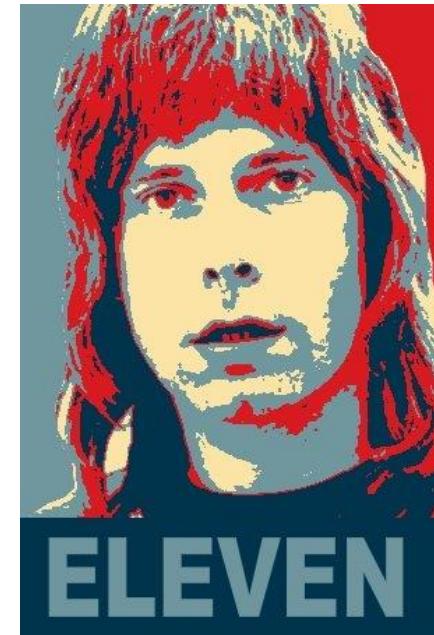
# Android 11



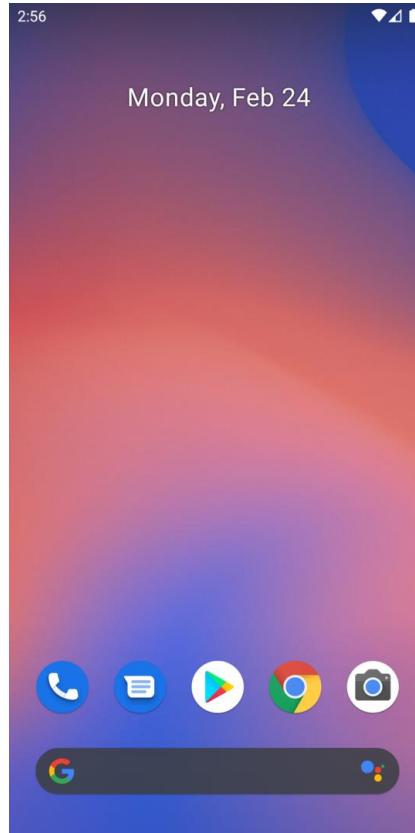
# Brief Aside...



“It’s one louder!”



# Android 11

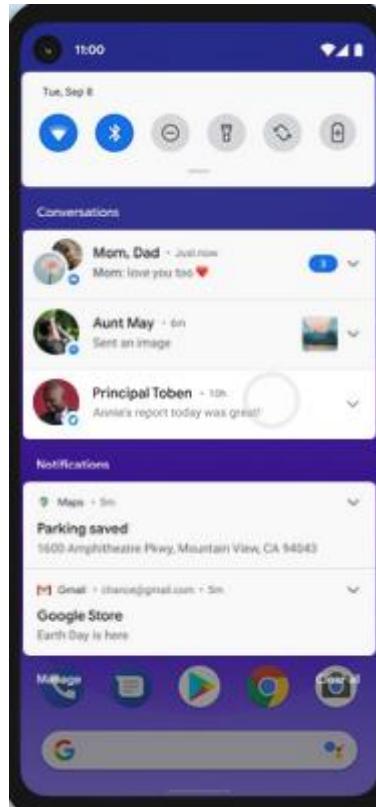


# Android 11

## Manage Conversations.

Get all your messages in one place.

See, respond to and control your conversations across multiple messaging apps. All in the same spot. Then select people you always chat with. These priority conversations show up on your lock screen. So you never miss anything important.

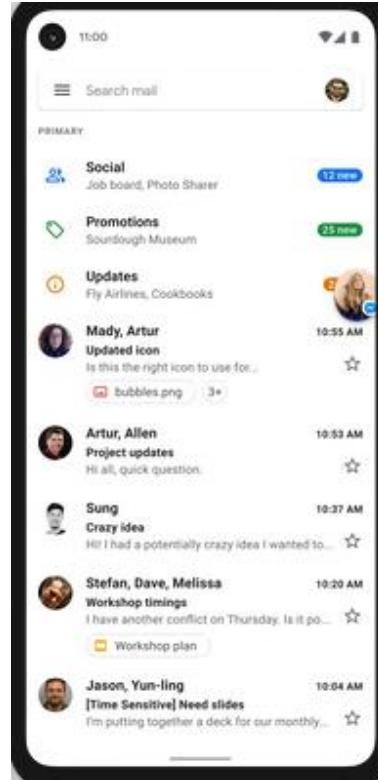


# Android 11

## Bubbles.

### Chat while multitasking.

With Android 11, you can pin conversations so they always appear on top of other apps and screens. Bubbles keep the conversation going – while you stay focused on whatever else you're doing. Access the chat anytime or anywhere. Then carry on doing you.

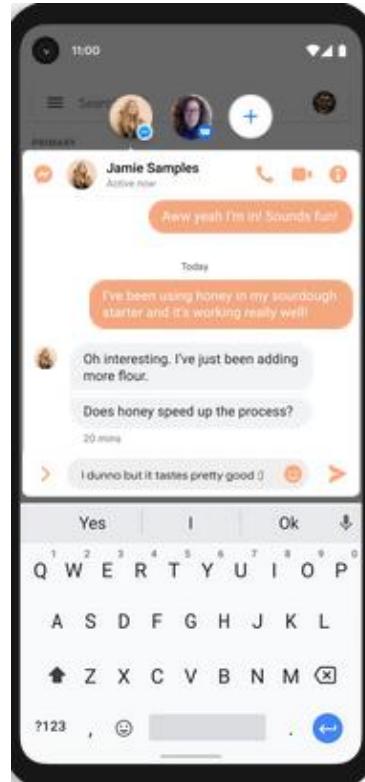


# Android 11

## Bubbles.

### Chat while multitasking.

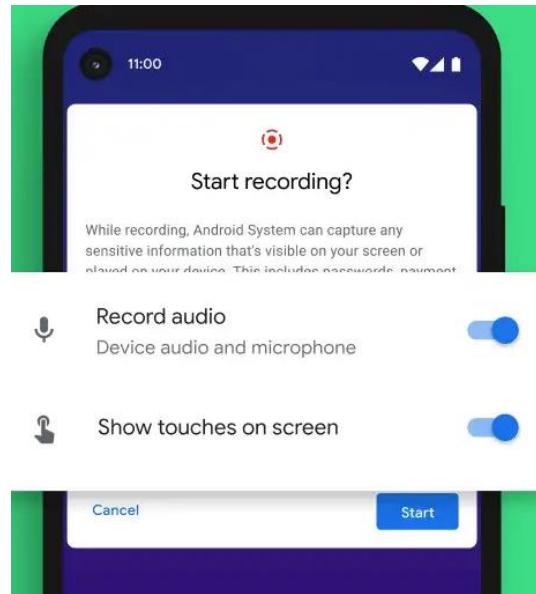
With Android 11, you can pin conversations so they always appear on top of other apps and screens. Bubbles keep the conversation going – while you stay focused on whatever else you're doing. Access the chat anytime or anywhere. Then carry on doing you.



# Android 11

## Built-in screen recording. Finally.

Screen recording lets you capture what's happening on your phone. And it's built right into Android 11, so you don't need an extra app. Record with sound from your mic, your device or both.



# Android 11

## Smart reply

Get suggested responses in conversations.

Hi there, sounds good or  when you need it.  
On Pixel devices, replies are intelligently suggested based on what's been said in the conversation – so the words you need and perfect emoji are always at your fingertips.

Hey, Jason, we are in the lobby.

Hey, I'm headed out now.

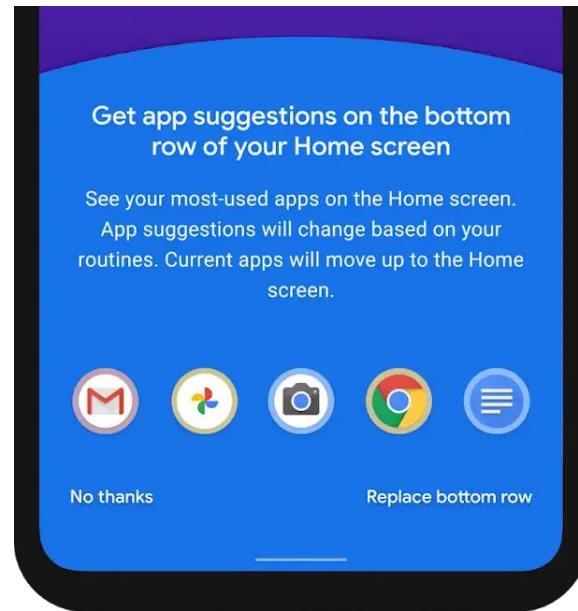
Sounds good!

# Android 11

## App suggestions

Easily get to apps you need most.

Pixel devices make app suggestions that change based on your routine. So you can see what you need throughout the day, all on the bottom of your home screen. Making it easy to get to your morning workout app when you wake up. Or to access the TV app you use when it's time to relax.

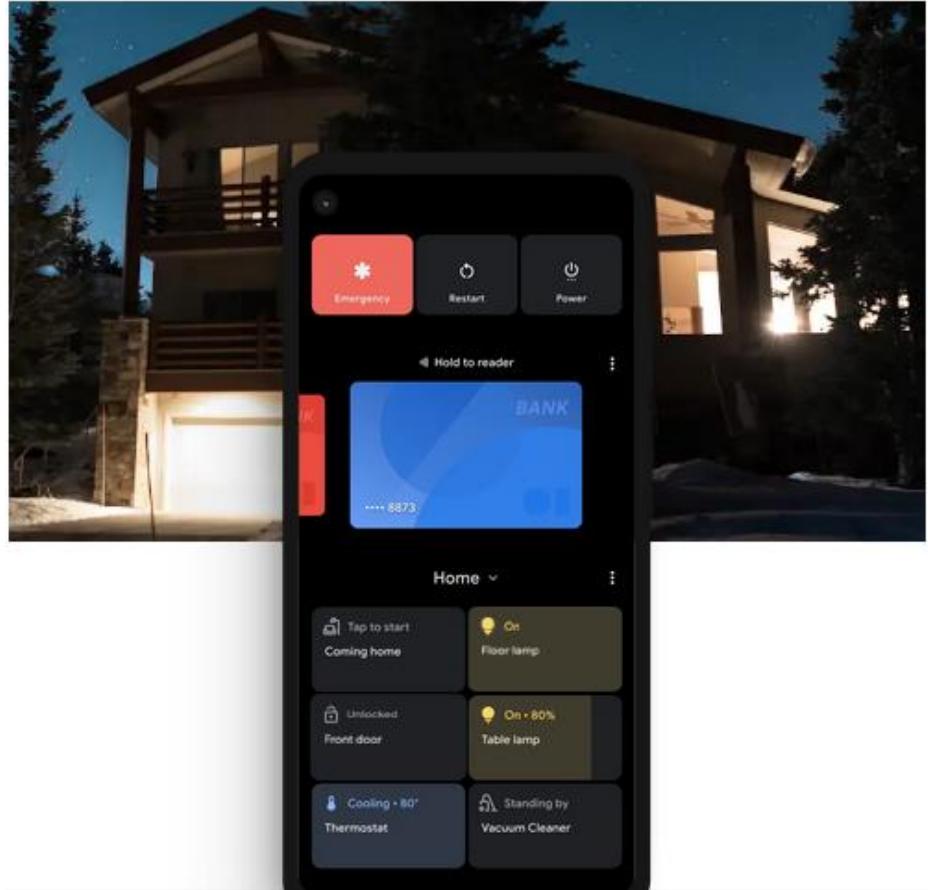


# Android 11

## Device Controls

Control your connected devices from one place.

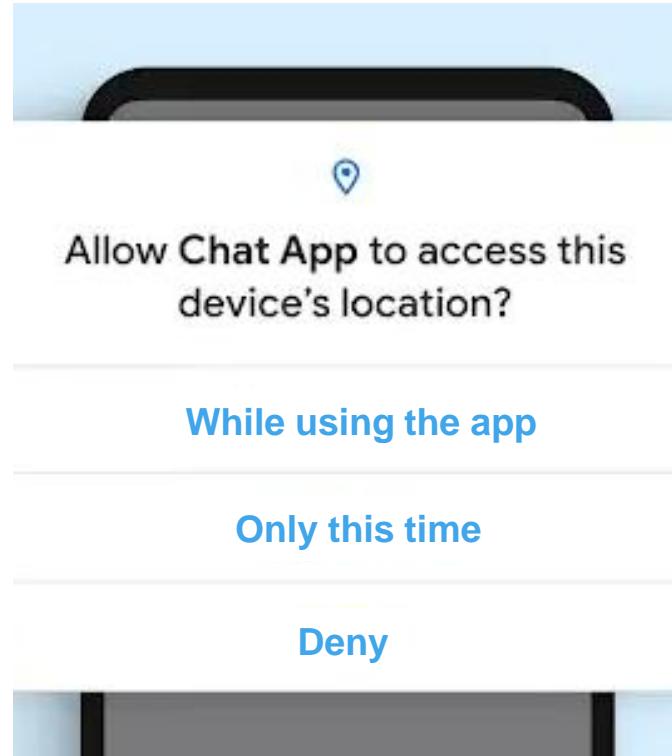
Set the temperature to chill, then dim your lights. All from a single spot on your phone. just long press the power button to see and manage your connected devices. Making life at home much easier.



# Android 11

## One-time permissions

Give one-time permissions to apps that need your mic, camera or location. The next time the app needs access, it must ask for permission again.



# Assessment

- **Exam 50%**
  - Changes foreseen i.e. structure of the paper different to last year.
  - Format likely Open Book.
  - **Must be passed to pass the module**

# Assessment

- **Exam** 50%
  - Changes foreseen i.e. structure of the paper different to last year.
  - Format likely Open Book.
  - **Must be passed to pass the module**
- **CA** 50%
  - 3 Lab tests (5% each), provisionally weeks 5, 9 and 12
  - 1 Assignment (35%) released **Reading Week** due end of Week 12
  - Demo of Assignment Week 13 **No demo, No grade, No exceptions.**

# Any Questions?

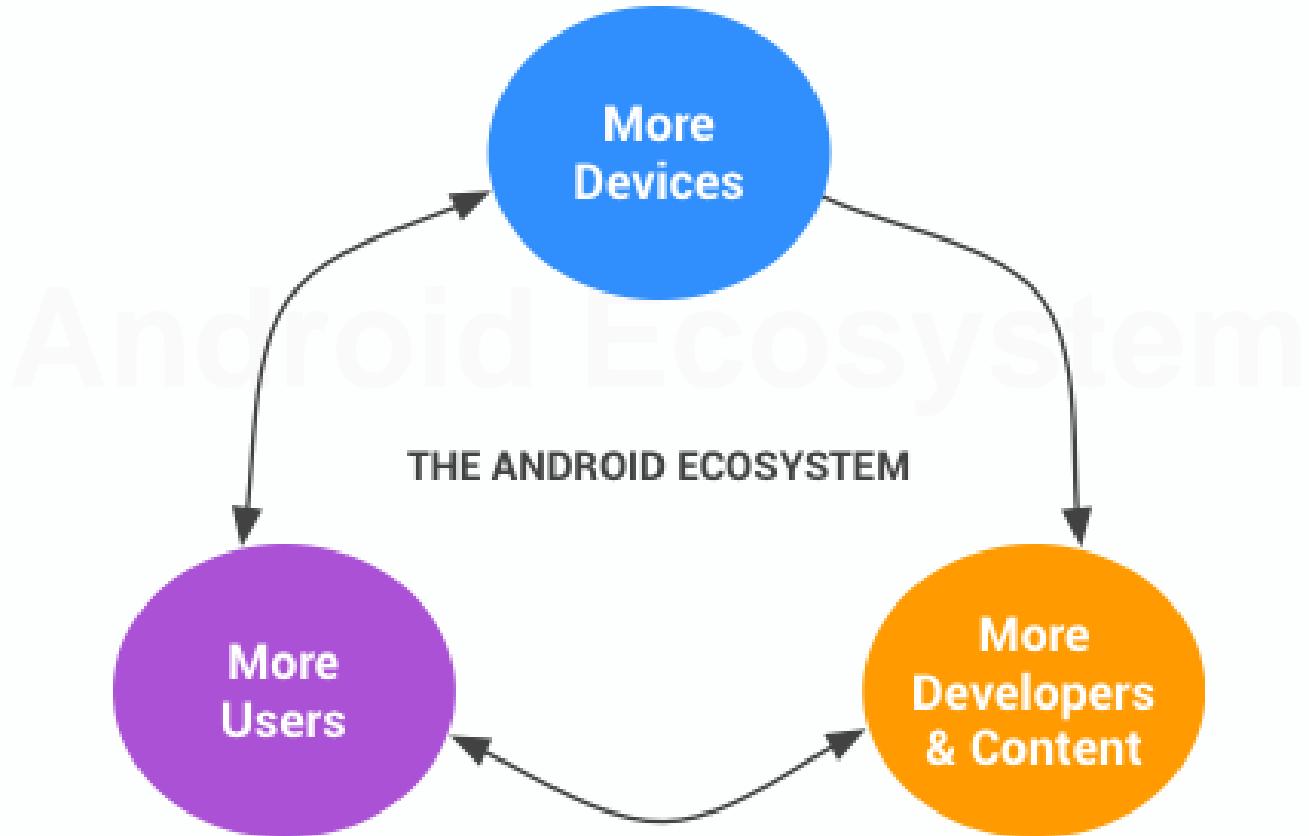


# Mobile Software Development

**CMPU 3036 DT211C/3, DT282/3**

# Today's Lecture

- Android Ecosystem
- Android Platform Architecture
- Android History
- Challenges of Android App Development
- App Fundamentals



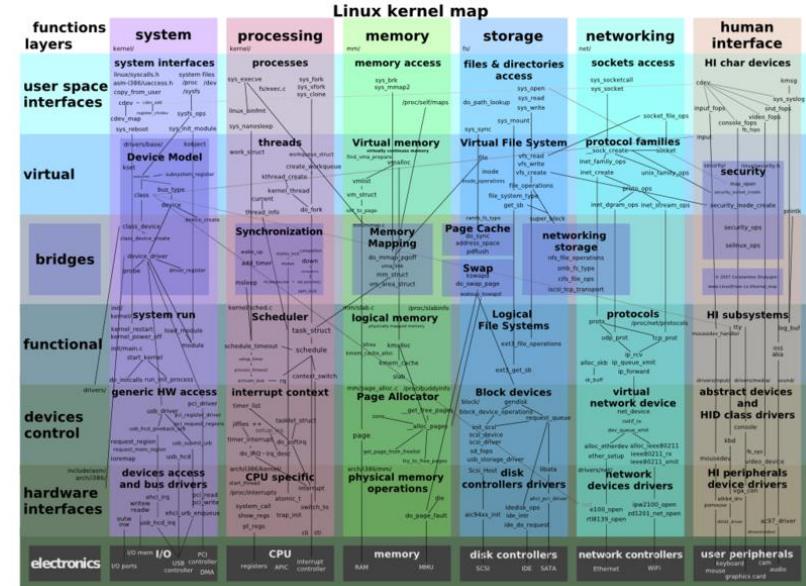
# What is Android?

- Mobile **Operating System** based on [Linux kernel](#)



# What is Android?

- Mobile Operating System based on Linux kernel



# What is Android?

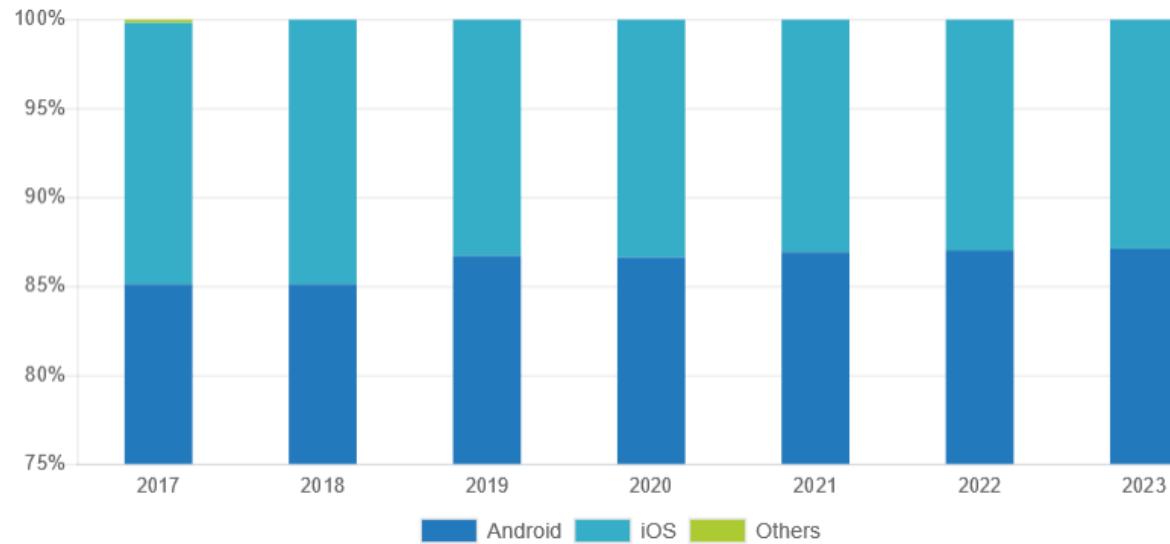
- User Interface for **Touch Screens**



# What is Android?

- Used on over 80% of all smartphones

Worldwide Smartphone Shipment OS Market Share Forecast



# What is Android?

- Powers devices such as cars, TVs, and watches.



# What is Android?

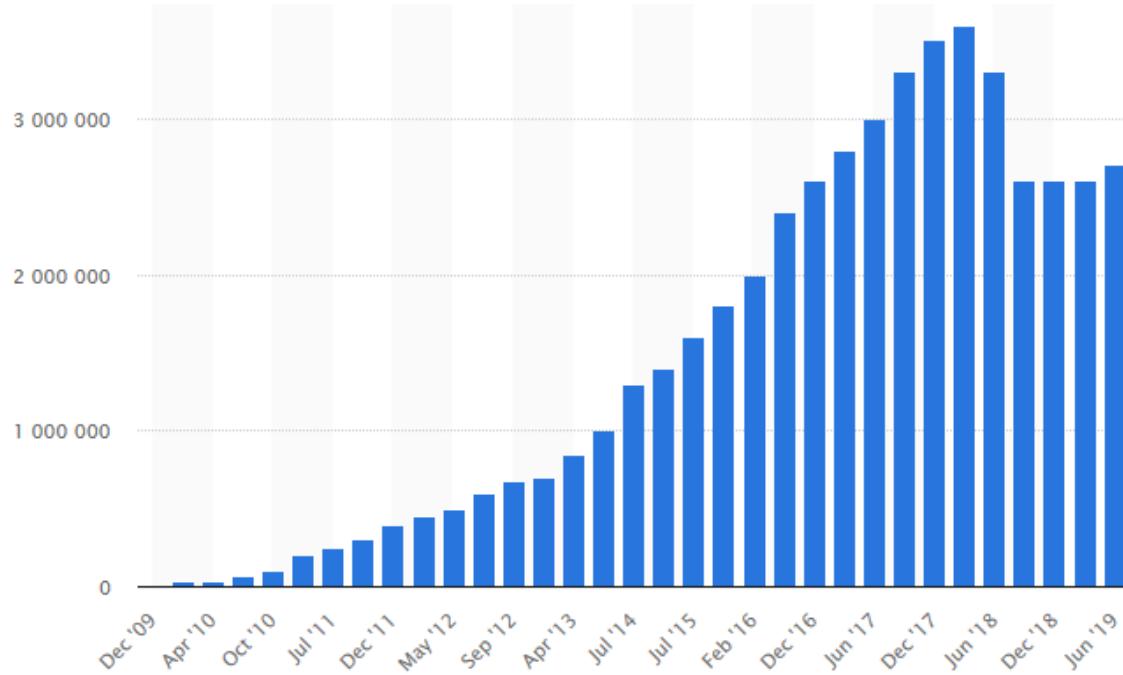
- Over 2 Million  
Android apps in  
Google Play  
store



# What is Android?

- Over 2 Million

Android apps in  
Google Play  
store



# What is Android?

- Highly **customisable** for devices / by vendors



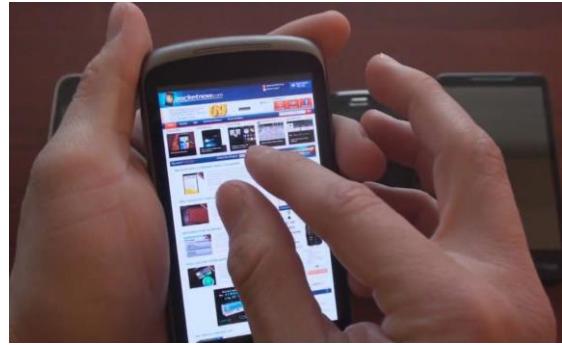
# What is Android?

- Open Source



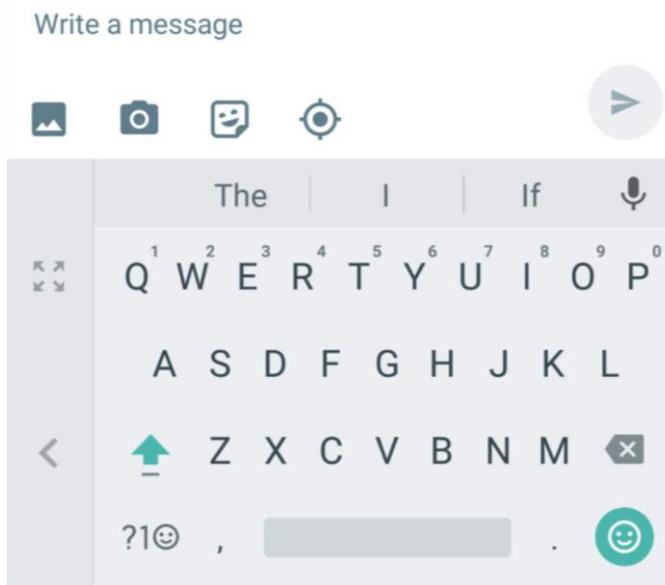
# Android User Interaction

- Touch **Gestures**: Tapping, Swiping and Pinching



# Android User Interaction

- **Virtual Keyboard** for characters, numbers, and emoji 😊



# Android User Interaction

- Support for Bluetooth, USB controllers and Peripherals



Connect other devices to your phone's Internet connection. Select connection method.

## USB tethering

Turn on USB tethering and connect other devices to share my data

OFF ON

## Wi-Fi hotspot

OFF ON

## Bluetooth tethering

Sharing Internet connection...

OFF ON

Help

# Android and Sensors

Sensors can **discover** User action and respond



Accelerometer



Gyroscope



Compass



GPS



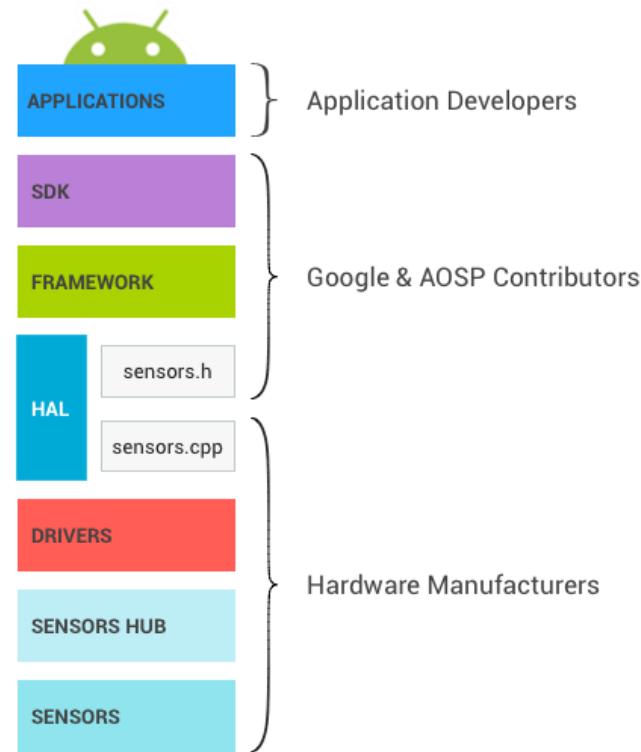
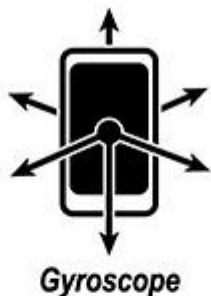
Light sensor



Barometer

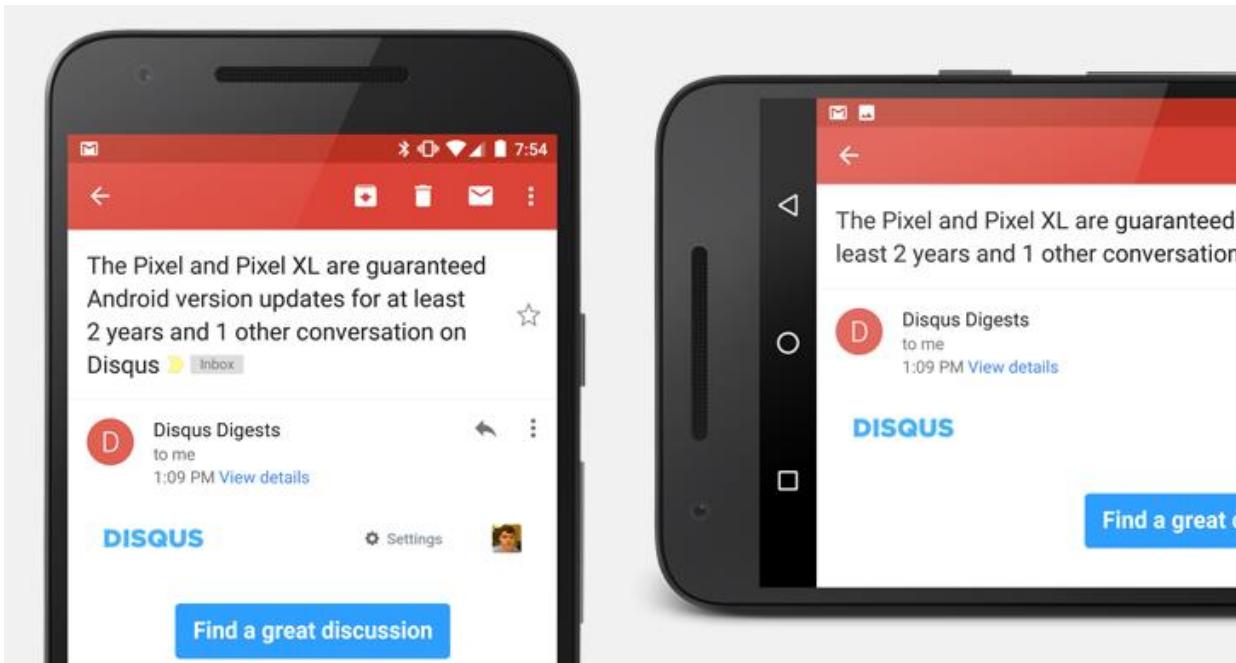
# Android and Sensors

Sensors can **discover** User action and respond



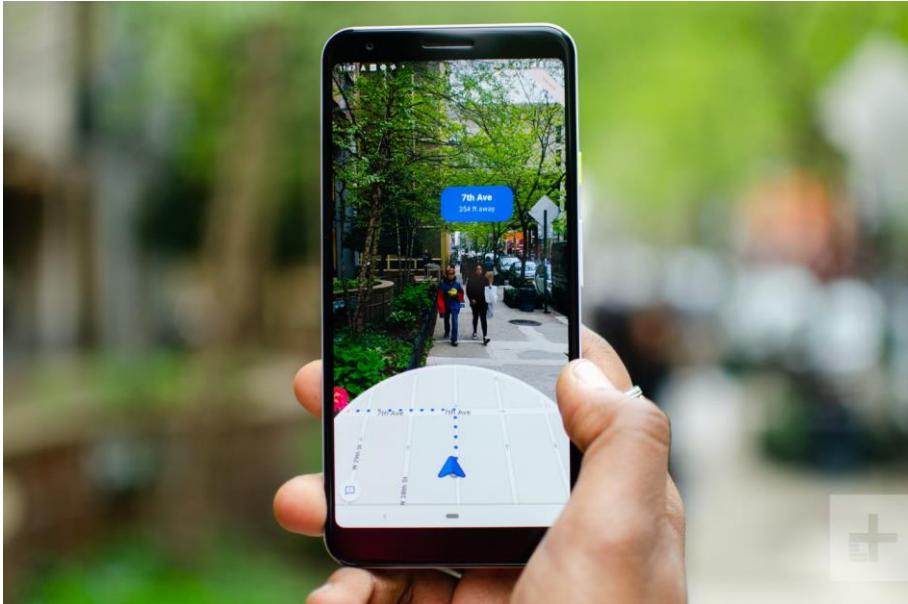
# Android and Sensors

- Device contents **rotate** as needed



# Android and Sensors

- Walking adjusts position on **Map**



# Android and Sensors

- **Tilting** steers a virtual car or **controls** a physical toy



# Android and Sensors

- Moving too fast disables game interactions

# Android Home Screen

- Launcher **icons** for apps



# Android Home Screen

- Self-updating **widgets** for live content



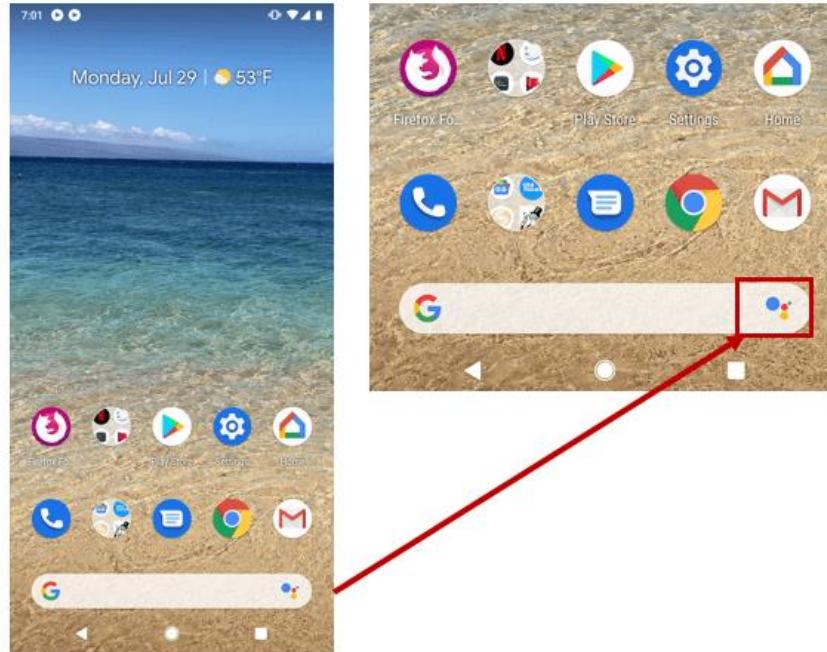
# Android Home Screen

- Folders to **organise** apps



# Android Home Screen

- **Voice Assistance**



# Rated Android Apps



Zedge 4.6



Waze 4.4



Speedtest 4.4



Xender 4.5



Unified Remote 4.6



TikTok 4.4



Alarmy 4.7



Todoist 4.7



Tiny Scanner 4.8

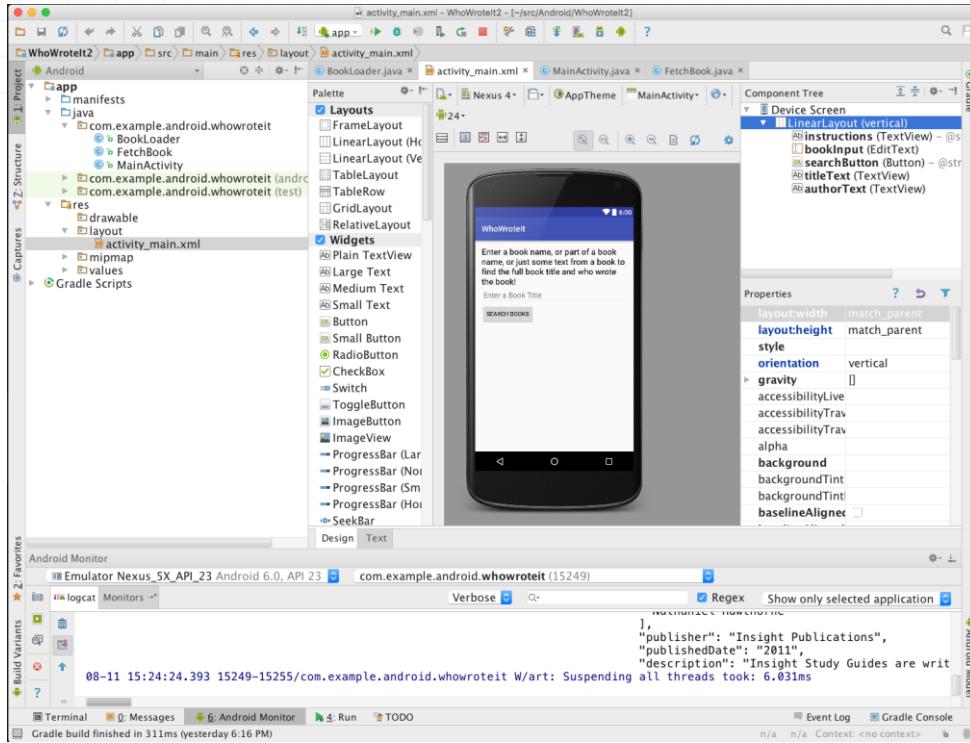


SwiftKey 4.5

# Android Software Developer Kit (SDK)

- Development Tools (debugger, monitors, editors)
- Libraries (maps, wearables)
- Virtual Devices (emulators)
- Documentation ([developers.android.com](http://developers.android.com))
- Sample Code

# Android Studio



- Official Android IDE
- Develop, run, debug, test, and package apps
- Monitors and performance tools
- Virtual devices
- Project views
- Visual Layout Editor

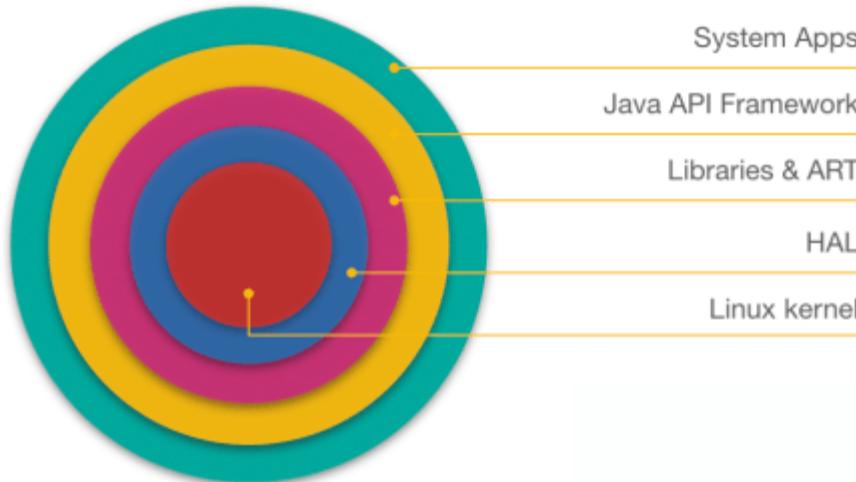


# Google Play Store

Publish apps through [Google Play](#) store:

- Official app store for Android
- Digital distribution service operated by Google

# Android Platform Architecture



# Android Stack

- At the bottom is the Linux Kernel.



Linux Kernel

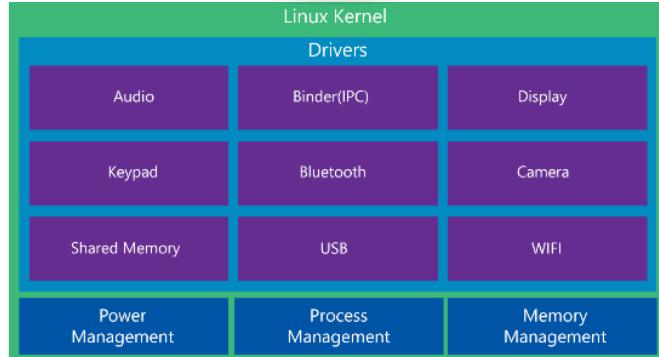
# Android Stack

- At the bottom is the Linux Kernel.
- The Android Runtime (ART) relies on it for **Threading & low-level Memory management**.
- Android also benefits from key **Security** features.



# Android Stack

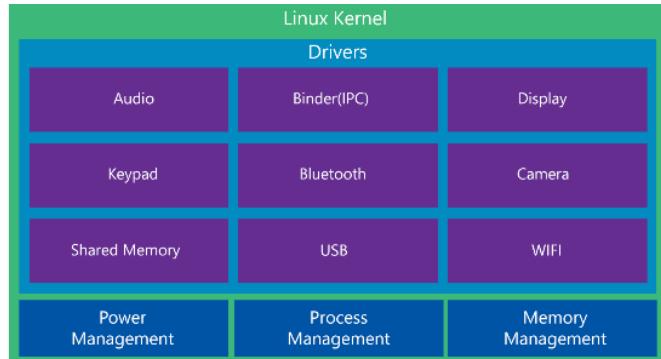
- At the bottom is the Linux Kernel.
- The Android Runtime (ART) relies on it for **Threading & low-level Memory management**.
- Android also benefits from key **Security** features.
- Device manufacturers can develop hardware drivers for a well-known kernel.



# Android Stack

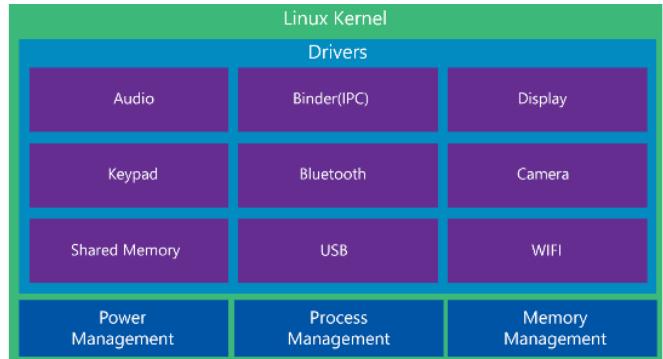
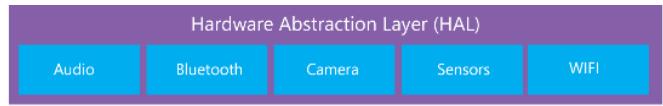
- The **Hardware Abstraction Layer (HAL)** provides standard interfaces that expose device hardware capabilities to the higher-level **Java API** framework.

Hardware Abstraction Layer (HAL)



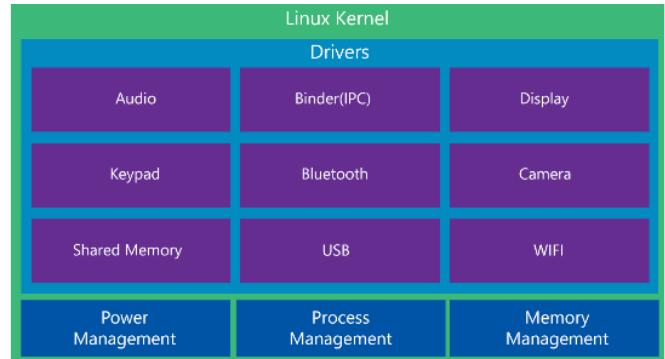
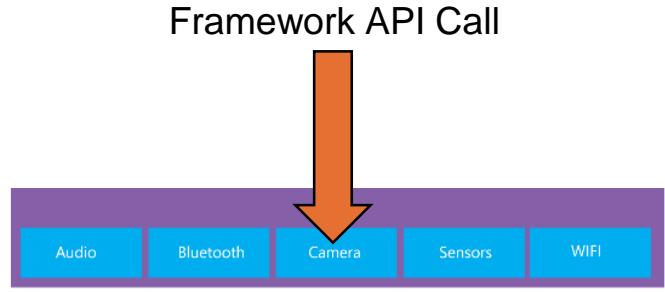
# Android Stack

- It consists of multiple **Library** modules, which implement an **interface** for a specific type of hardware component e.g. Camera or Bluetooth.



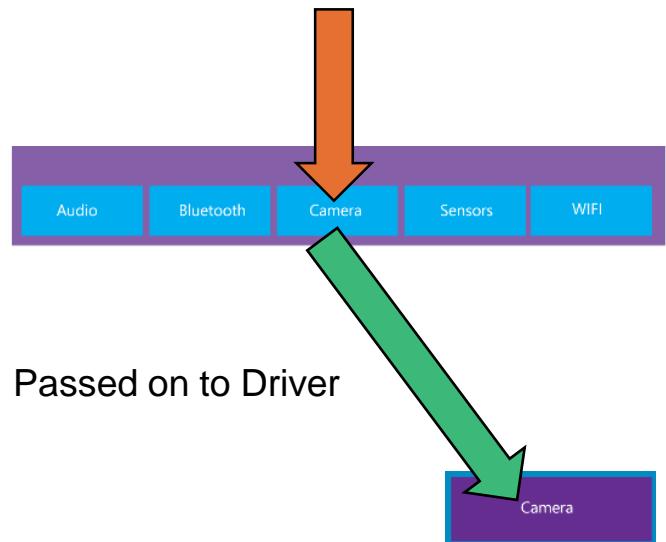
# Android Stack

- When a framework API makes a call to access device hardware, the **Android System** loads the **Library** module for that hardware component.



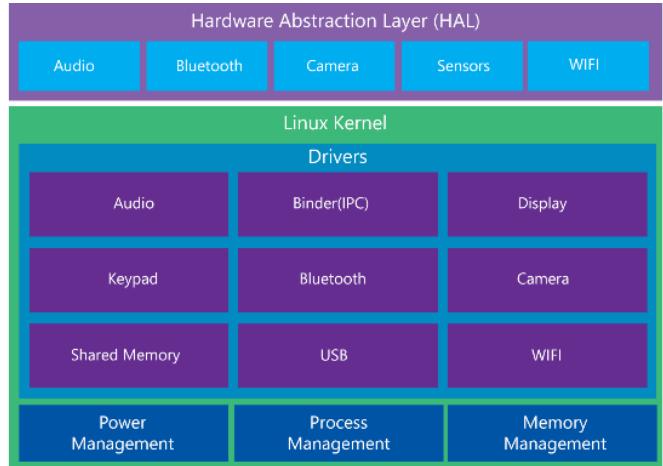
# Android Stack

- When a framework API makes a call to access device hardware, the **Android System** loads the **Library** module for that hardware component.



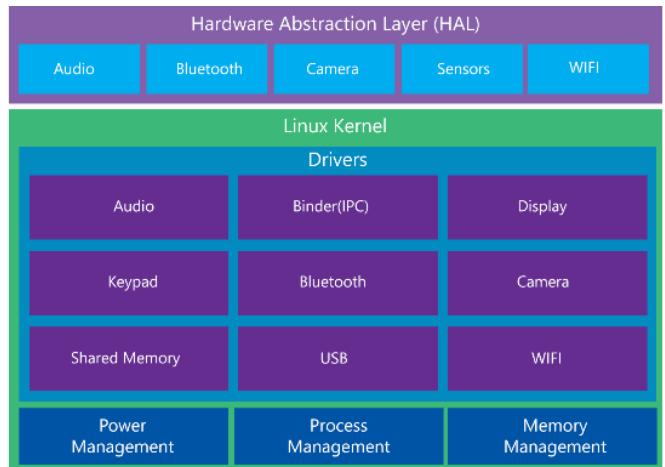
# Android Stack

- Prior to Android version 5.0 (API level 21),  
Dalvik was the Android runtime.



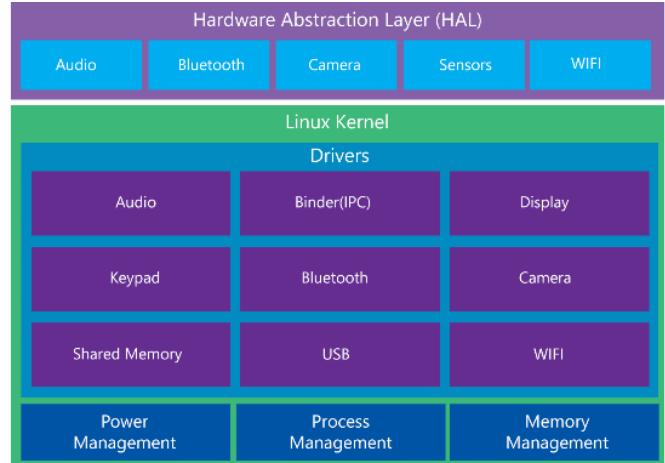
# Android Stack

- Prior to Android version 5.0 (API level 21),  
Dalvik was the Android runtime.
- Dalvik is the work of Virtual-Machine wrangler, Dan Bornstein.



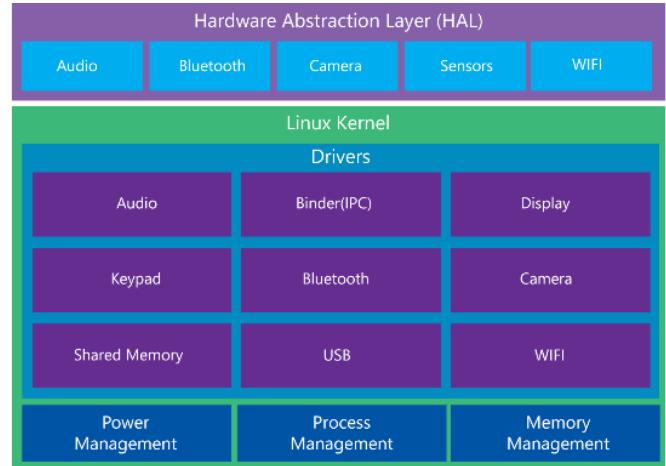
# Android Stack

- For devices running Android version 5.0 (API level 21) or higher each app runs in its own **process** and with its own instance of the Android Runtime (ART).



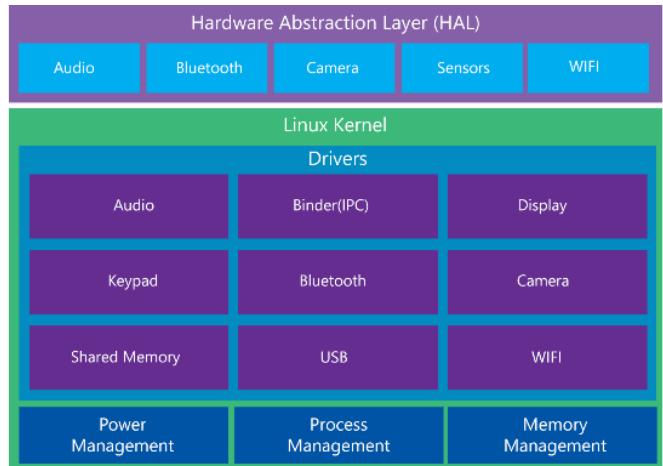
# Android Stack

- ART is written to run multiple virtual machines on low-memory devices by executing DEX files.



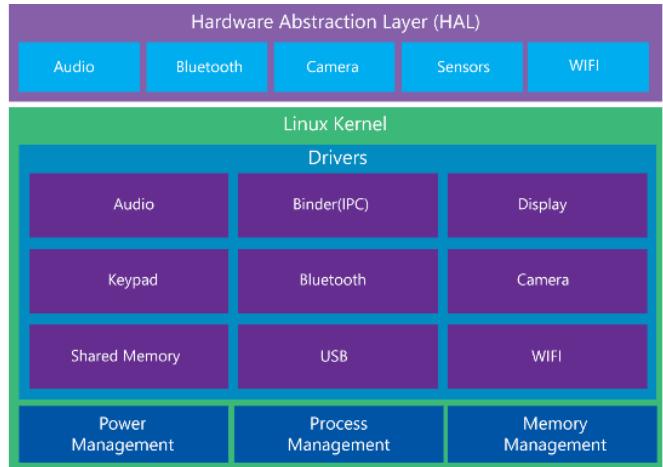
# Android Stack

- ART is written to run multiple virtual machines on low-memory devices by executing DEX files.
- DEX files are a bytecode format designed specially for Android that's **optimised** for minimal memory footprint.



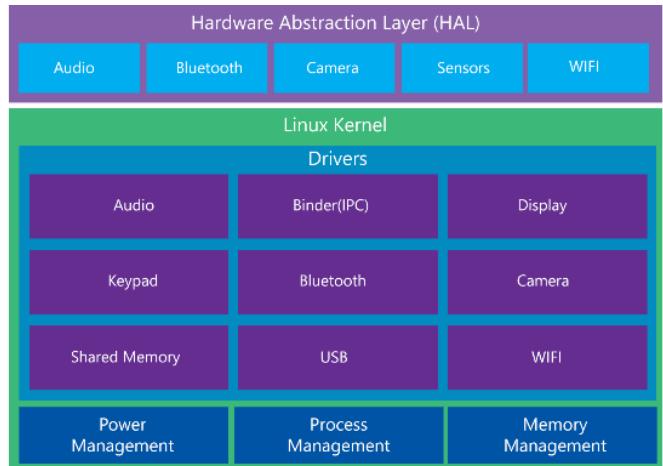
# Android Stack

- **Build toolchains** **compile** Java sources into DEX bytecode, which can run on the Android platform.



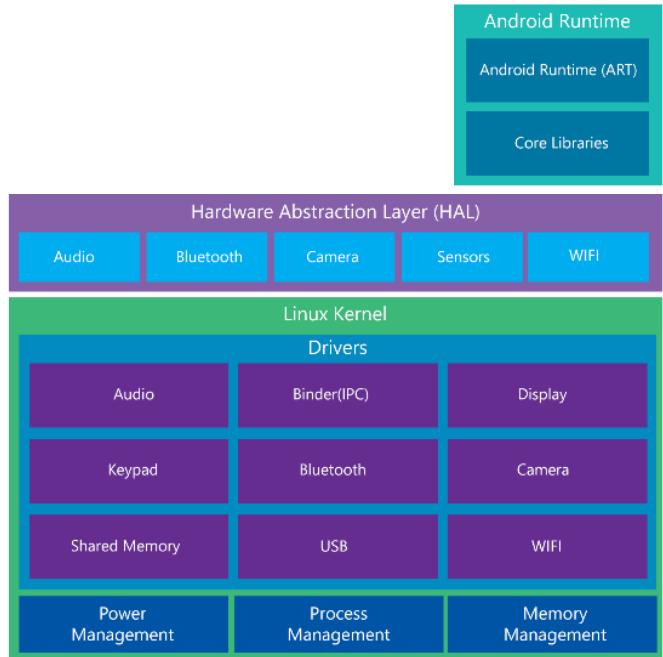
# Android Stack

- Build toolchains **compile** Java sources into DEX bytecode, which can run on the Android platform.
- If your app runs well on ART, then it should work on Dalvik as well, but the reverse may not be true ( $\leftarrow$  backwards compatible).



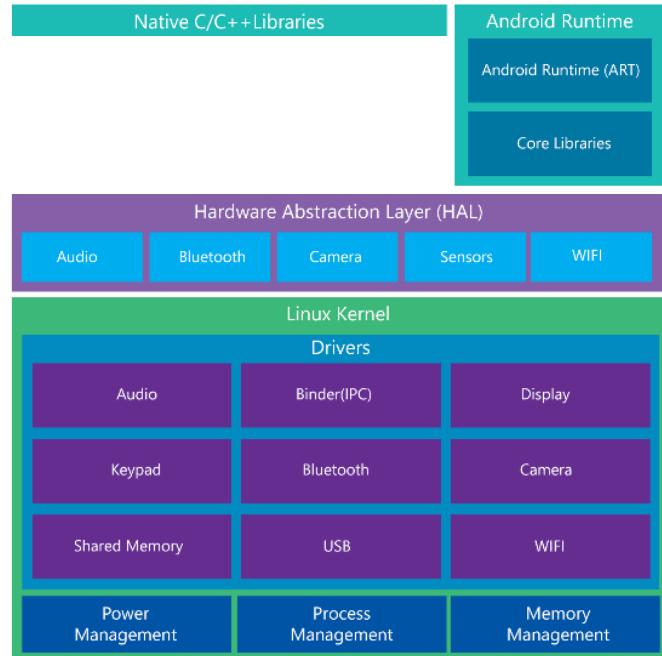
# Android Stack

- Android also includes a set of core runtime libraries that provide most of the functionality of the Java Programming Language.



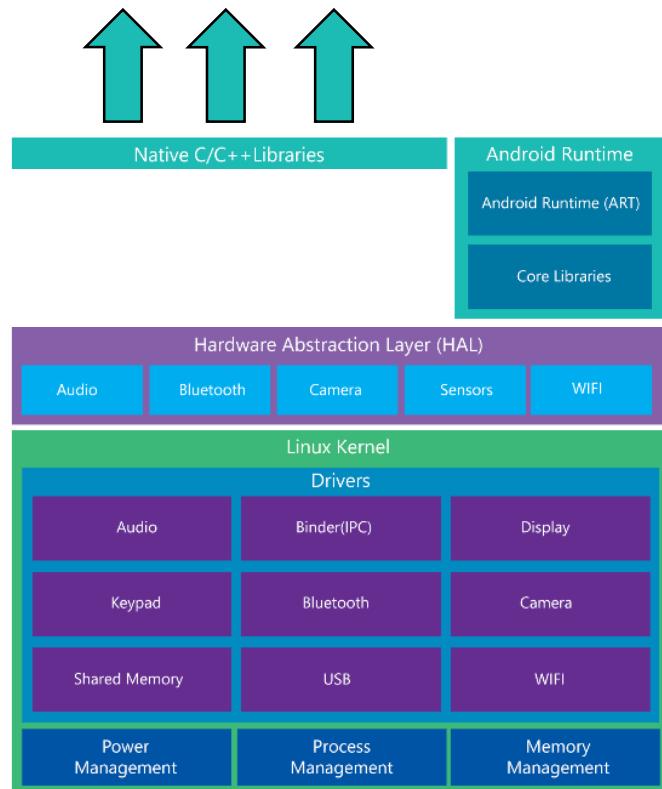
# Android Stack

- Many core Android system components and services, such as ART and HAL, are built from **native code** that require libraries written in C and C++.



# Android Stack

- The Android platform provides Java framework APIs to expose the functionality of some of these native libraries to apps.



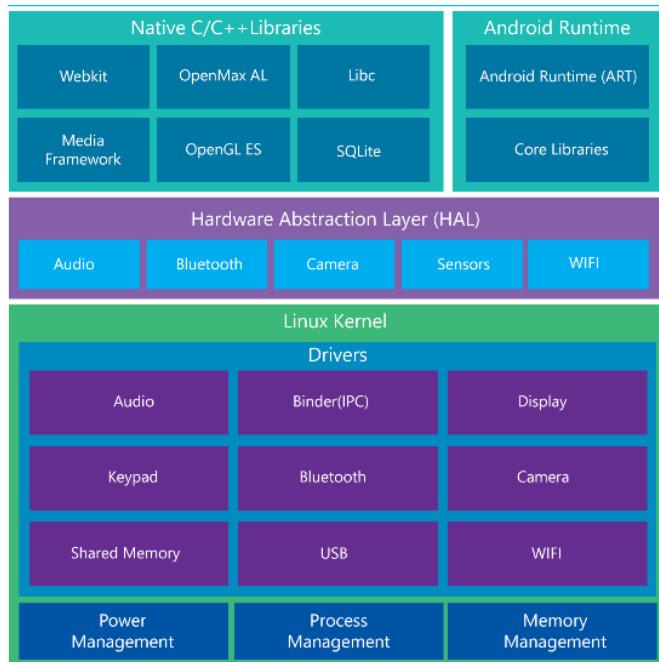
# Android Stack

- **OpenGL ES**
  - Adds support for drawing and manipulating 2D and 3D graphics
- **Wekit**
  - Browser engine to display HTML
- **Media Framework**
  - Playback, recording e.g. MP3, AAC, H.264, PNG.



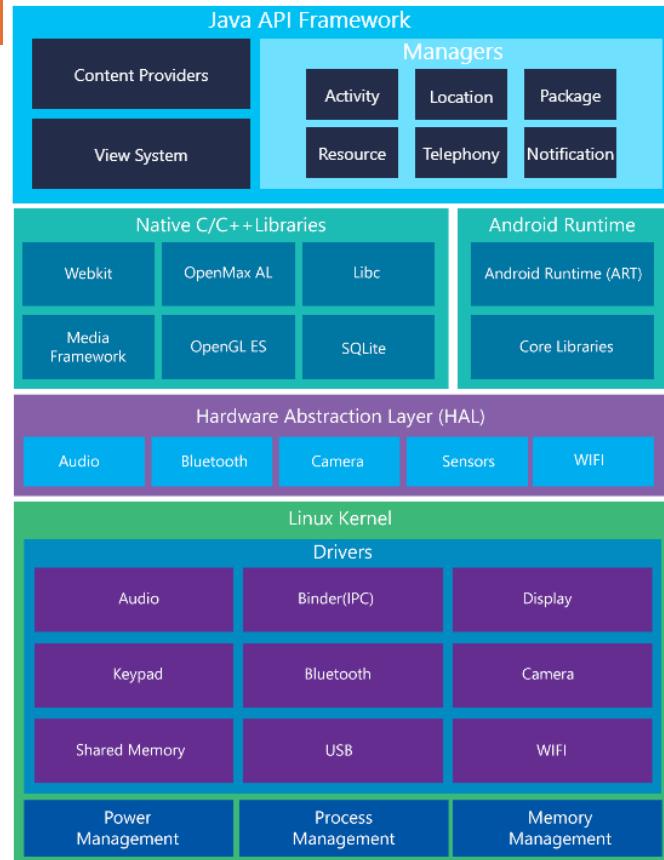
# Android Stack

- If developing an app that requires C or C++ code, you can use the Android NDK to access some of these native platform libraries **directly** from your code.



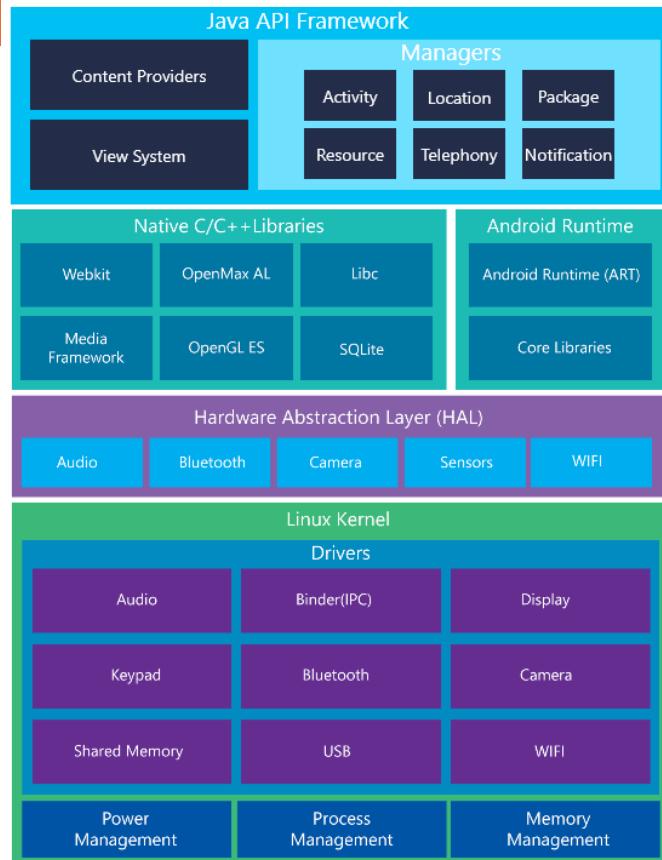
# Android Stack

- The entire feature-set of the Android OS is available to you through APIs written in the **Java** language.



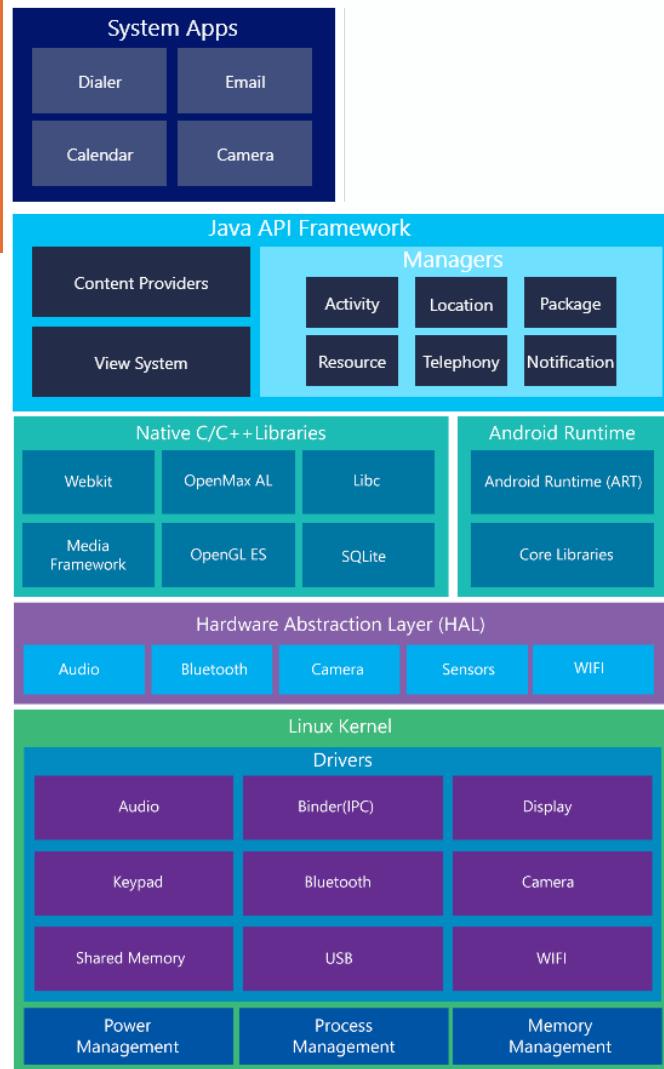
# Android Stack

- The entire feature-set of the Android OS is available to you through APIs written in the **Java** language.
- These APIs form the building blocks you need to create Android apps by simplifying the **reuse** of key system components and services.



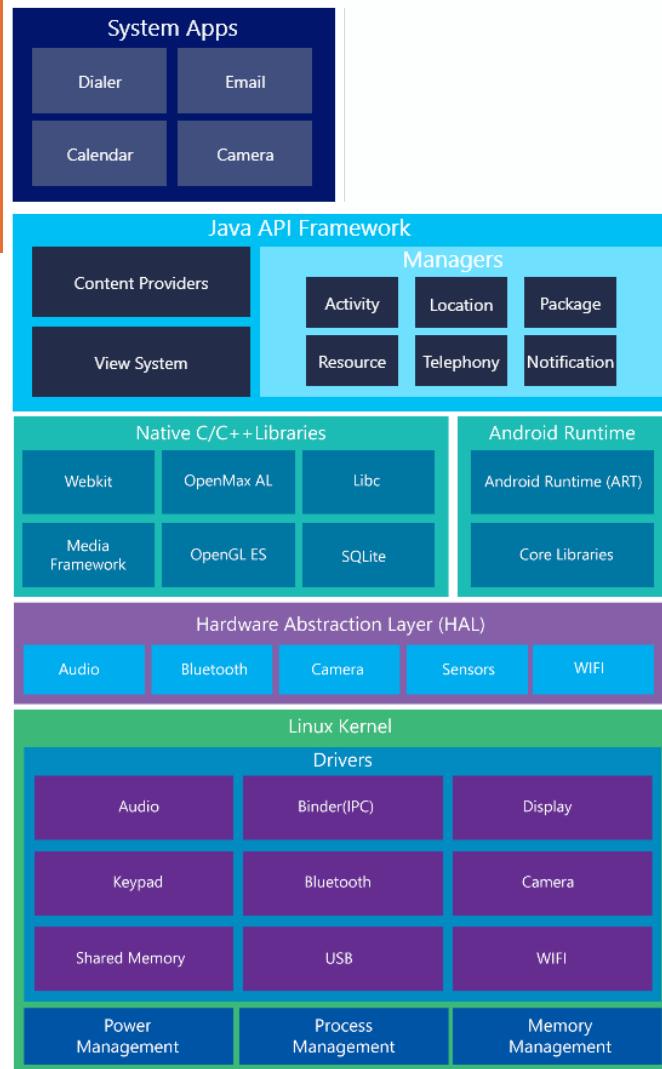
# Android Stack

- Android comes with a set of core apps for **email**, SMS messaging, calendars, internet browsing, contacts, and more.



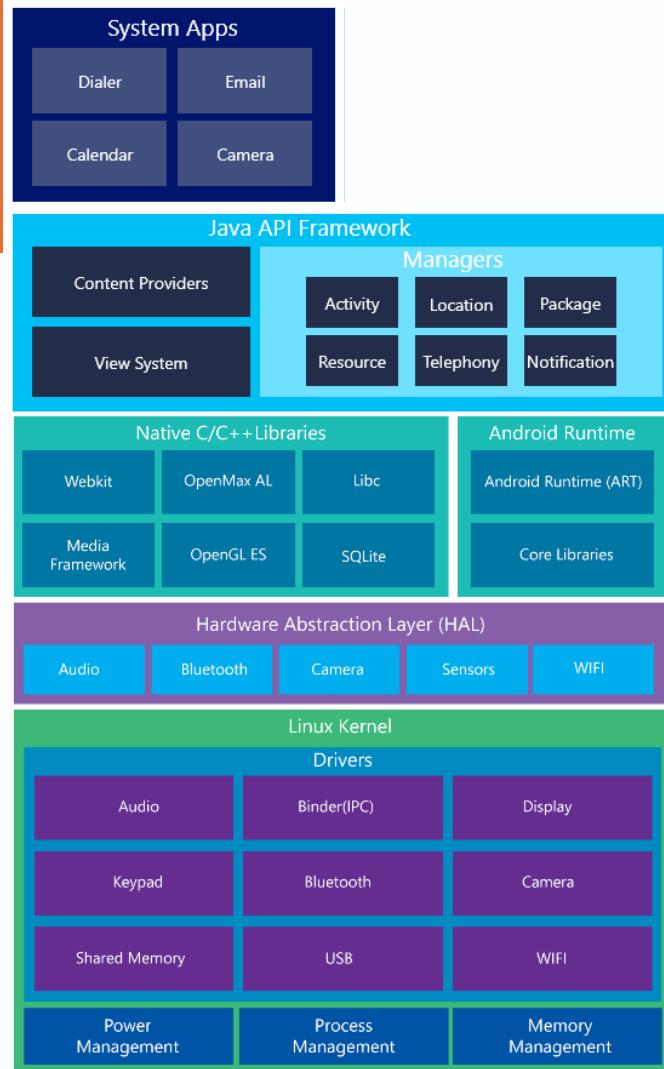
# Android Stack

- Android comes with a set of core apps for **email**, SMS messaging, calendars, internet browsing, contacts, and more.
- Apps included with the platform have no special status among the apps the user chooses to install.



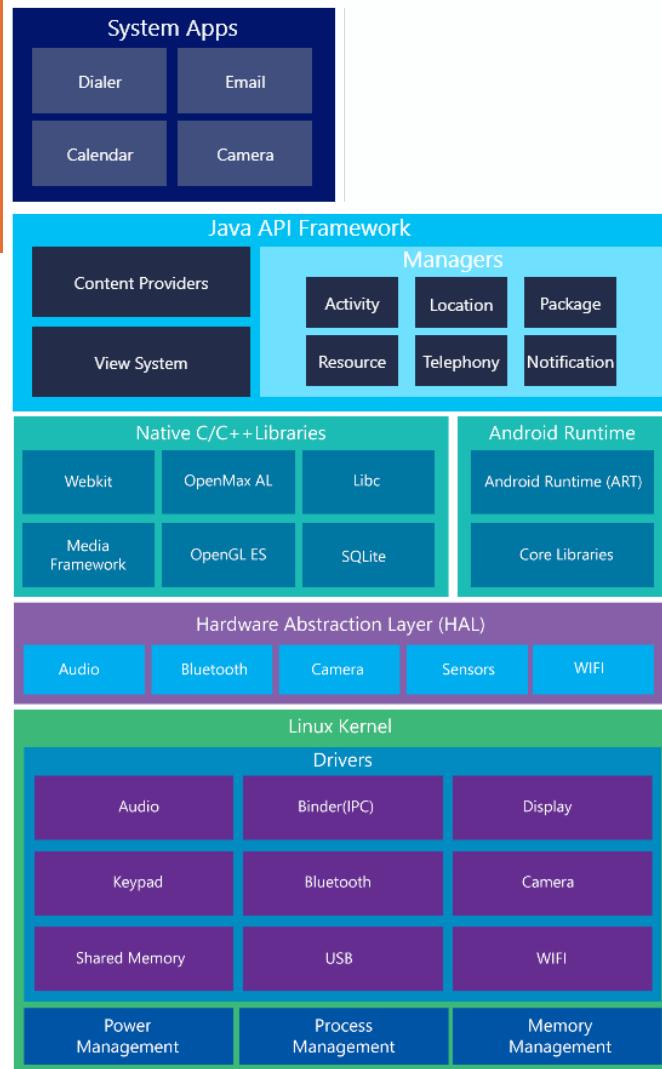
# Android Stack

- So a **third-party** app can become the user's default web browser, SMS messenger etc..



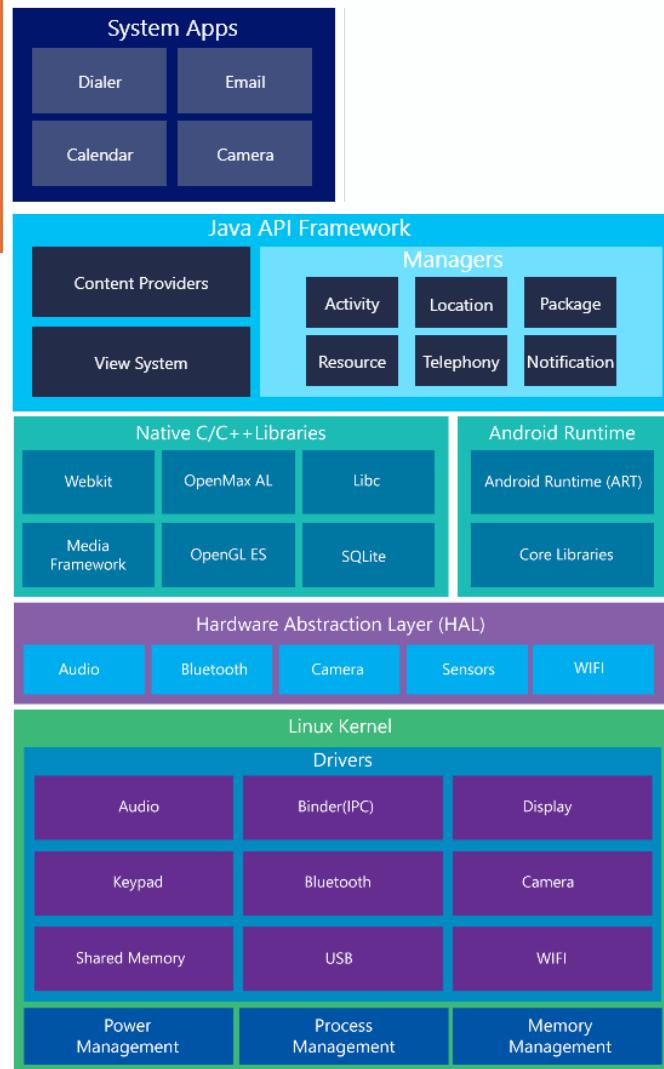
# Android Stack

- The system apps function both as apps for users **and** to provide key capabilities that developers can access from their own app.



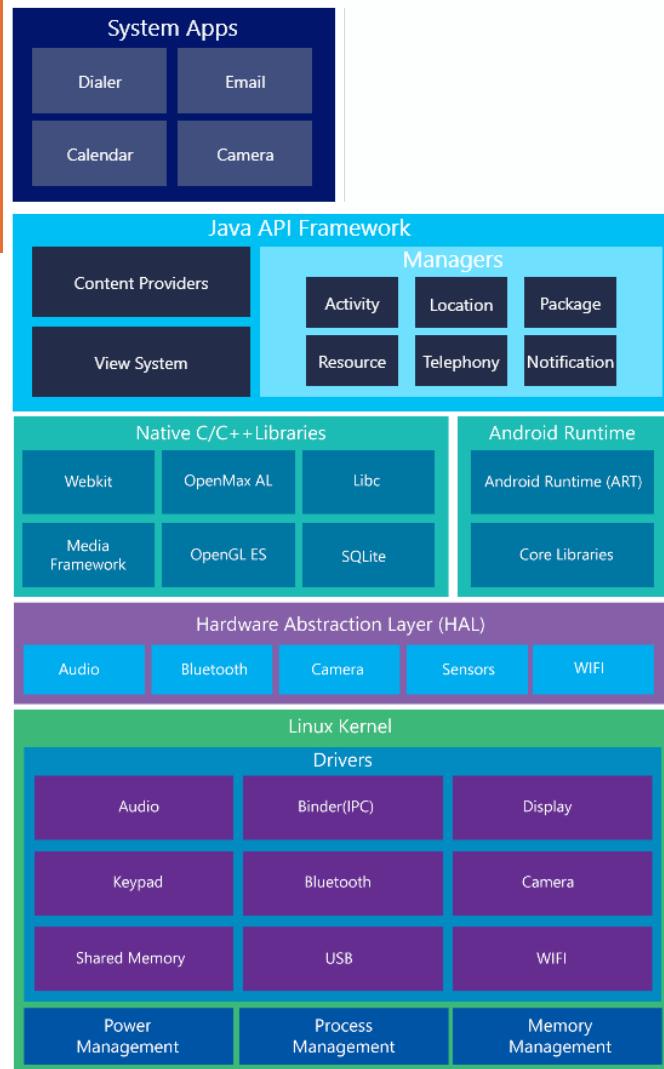
# Android Stack

- The system apps function both as apps for users **and** to provide key capabilities that developers can access from their own app.
- If your app would like to deliver an SMS message, you don't need to build that functionality **yourself** - you can instead call whichever SMS app is already installed to deliver a message.

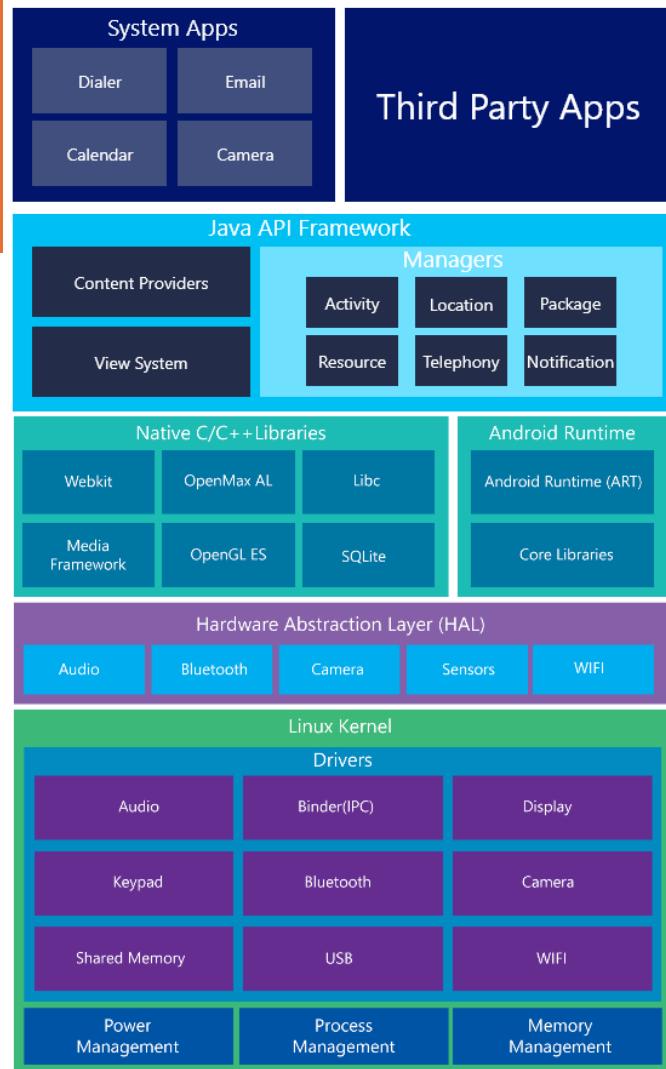


# Android Stack

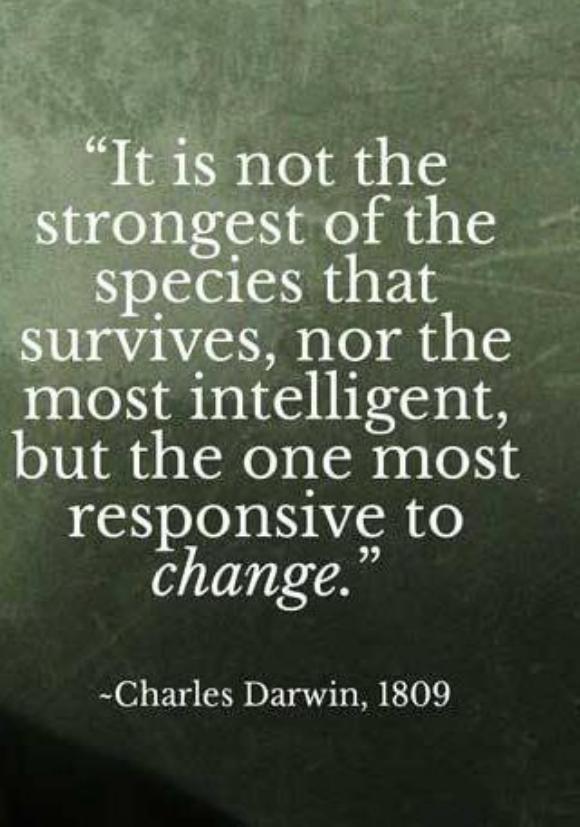
- Developers also have full access to the same framework APIs that Android system apps use.



# Android Stack



# History Lesson



“It is not the strongest of the species that survives, nor the most intelligent, but the one most responsive to *change*.”

~Charles Darwin, 1809

# Pre-History



Cupcake

2009



- Support for third-party virtual keyboards with text prediction.
- Video recording and playback (MPEG-4 and 3GP formats)

# Pre-History



Cupcake  
2009



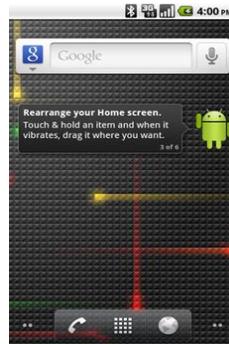
Donut  
2009



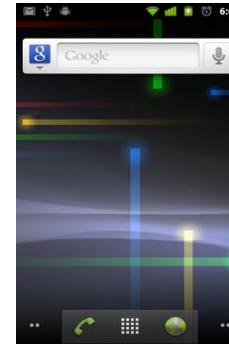
Eclair  
2009



Froyo  
2010

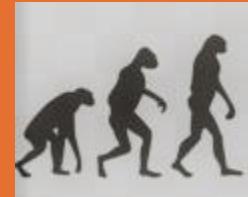


GingerBread  
2010



- Updated user interface design with increased simplicity and speed
- Support for NFC, allowing the user to read an NFC tag embedded in a poster

# Middle Age

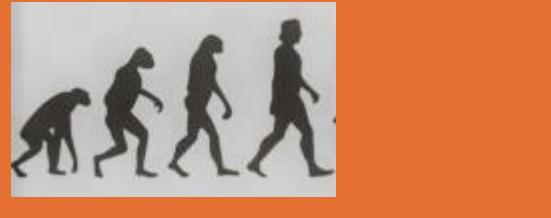


HoneyComb  
2011



- Support for multi-core processors, ability to encrypt all user data
- Multiple browser tabs replacing browser windows and a new “incognito” mode

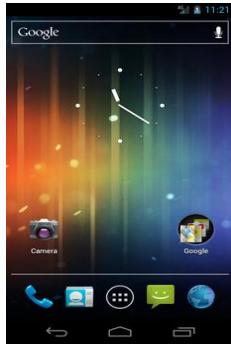
# Middle Age



HoneyComb  
2011



IceCream Sandwich  
2011



JellyBean  
2012



KitKat  
2012

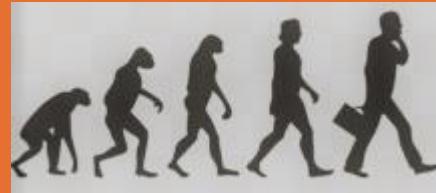


Lollipop  
2014



- Android Runtime (ART) with ahead-of-time (AOT) compilation and improved GC
- Support for 64-bit CPUs, **Material Design**, bringing a restyled User Interface

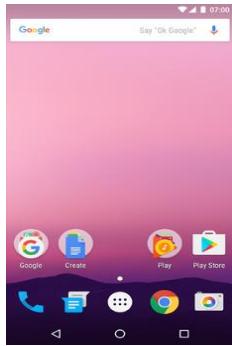
# Modern Era



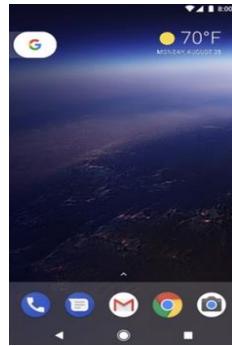
MarshMallow  
2015



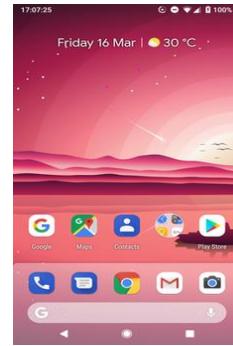
Nougat  
2016



Oreo  
2017

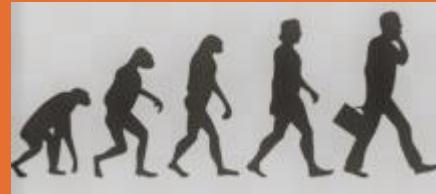


Pie  
2018



- Android Runtime (ART) with ahead-of-time (AOT) compilation and improved GC
- Support for 64-bit CPUs, **Material Design**, bringing a restyled User Interface

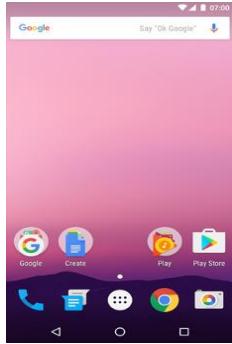
# Modern Era



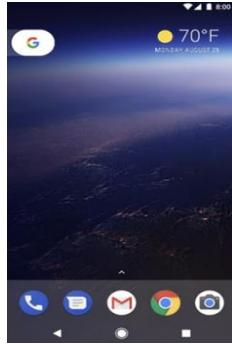
MarshMallow  
2015



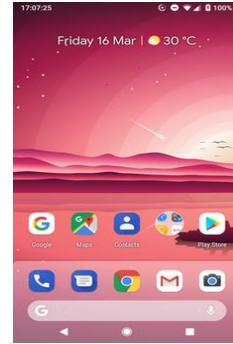
Nougat  
2016



Oreo  
2017

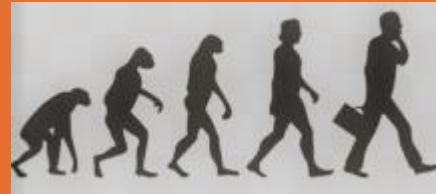


Pie  
2018



- Wind Down option lets users set a specific bedtime that enables Do Not Disturb
- Full conversation can be had within a notification

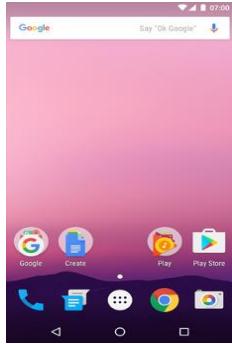
# Modern Era



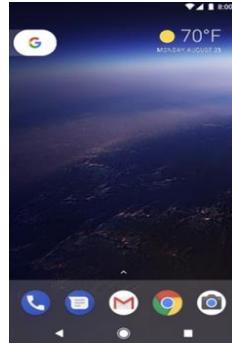
MarshMallow  
2015



Nougat  
2016



Oreo  
2017

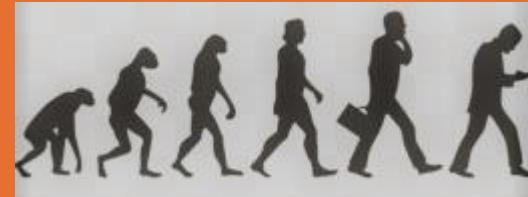


Pie  
2018



- "Lockdown" mode which disables biometric authentication once activated
- Rounded corners across the UI

# Today



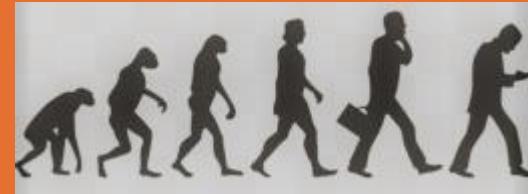
Android 10



- Platform optimisations have been made for foldable smartphones.
- A feature known as "bubbles" that presents content in pop-up overlays.

0

# Today



Android 10



- Native support added for MIDI controllers, AV1 video codec, Opus audio codec and HDR10
- A system-level dark theme.



# ANDROID APP DEVELOPMENT

# What is an Android App?

- One or more interactive screens.
- Written using [Java Programming Language](#) and [XML](#)
- **Kotlin** is a newer language for writing apps.
- Uses the Android Software Development Kit (SDK).
- Uses Android libraries and Android Application Framework.
- Executed by Android Runtime Virtual machine (ART).

# Challenges of Development

- Multiple screen sizes and resolutions.
- Performance: make your apps responsive and smooth.
- Security: keep source code and user data safe.
- Compatibility: run well on older platform versions.
- Marketing: understand the market and your users.  
(Hint: It doesn't have to be expensive, but it can be.)

# App Building Blocks

- Resources: layouts, images, strings, colors as XML and media files.
- Components: activities, services, and helper classes as Java code.
- Manifest: information about app for the runtime.
- Build configuration: APK versions in Gradle config files.

# Learn More

- [Android History](#)
- [Introduction to Android](#)
- [Platform Architecture](#)
- [UI Overview](#)
- [Platform Versions](#)
- [Supporting Different Platform Versions](#)
- [Android Studio User's Guide](#)

# Any Questions?



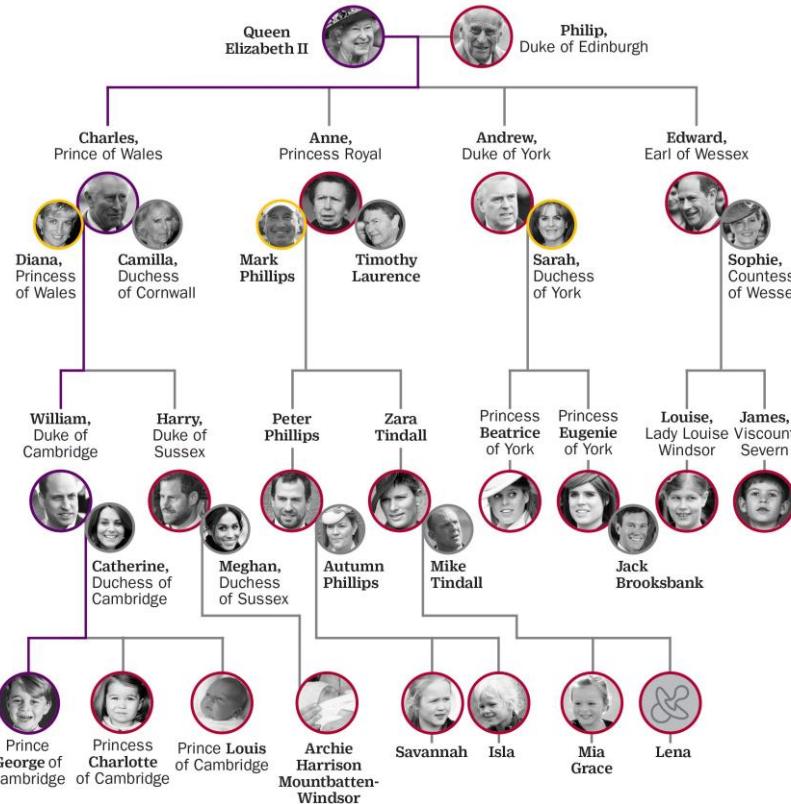
# Mobile Software Development

**CMPU 3036 DT211C/3, DT282/3**

# Today's Lecture

- Inheritance Recap
- Views, View groups, and View hierarchy
- Resources and Measurements

# Inheritance Recap



# Making a Game

```
public class Dude {  
    public String name;  
    public int hp = 100;  
    public int mp = 0;  
  
    public void sayName() {  
        System.out.println(name);  
    }  
    public void punchFace(Dude target) {  
        target.hp -= 10;  
    }  
}
```

# Need a Wizard Too

```
public class Wizard {  
    // ugh, gotta copy and paste  
    // Dude's stuff  
}
```

# Need a Wizard Too

But wait...

A Wizard does and has everything  
a Dude does and has!

# Need a Wizard Too

Don't Act Now!

You don't have to copy and paste!

# Subclass

```
public class Wizard extends Dude{  
}
```

# Buy Inheritance!

- Wizard can use everything\* the Dude has!

```
wizard1.hp += 1;
```

\*except for private fields  
and methods

- Wizard can do everything\* Dude can do!

```
wizard1.punchFace(dude1);
```

- You can use a Wizard like a Dude too!

```
dude1.punchFace(wizard1);
```

# Add Wizard Stuff

```
public class Wizard extends Dude{  
  
    ArrayList<Spell> spells;  
  
    public class cast(String spell) {  
        // cool stuff here  
  
        ...  
  
        mp -= 10;  
    }  
}
```

# Need a GrandWizard

```
public class GrandWizard extends Wizard{  
  
    public void sayName() {  
        System.out.println("Grand wizard" + name)  
    }  
}  
  
grandWizard1.name = "Flash"  
grandWizard1.sayName();  
((Dude)grandWizard1).sayName();
```

# What Java Does

```
grandWizard1.punchFace(dude1);
```

# What Java Does

```
grandWizard1.punchFace(dude1);
```

Look for punchFace in GrandWizard class

GrandWizard

# What Java Does

```
grandWizard1.punchFace(dude1);
```

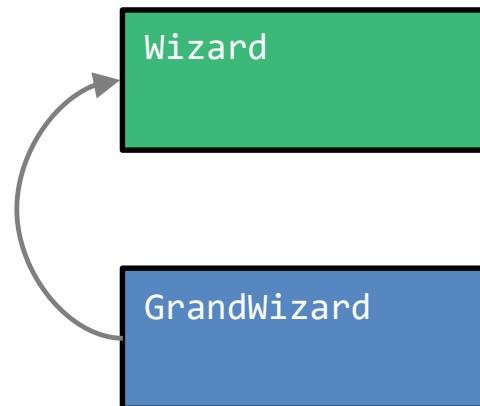
GrandWizard

It's not there! Does GrandWizard have a parent?

# What Java Does

```
grandWizard1.punchFace(dude1);
```

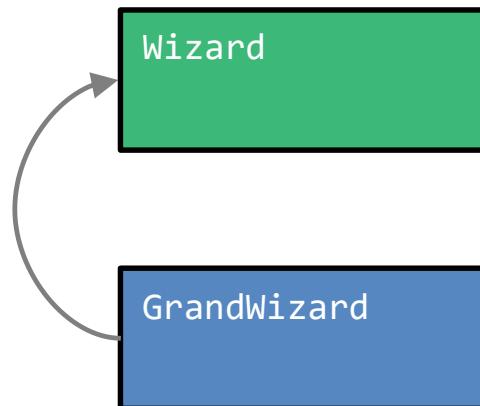
Look for punchFace in Wizard class



# What Java Does

```
grandWizard1.punchFace(dude1);
```

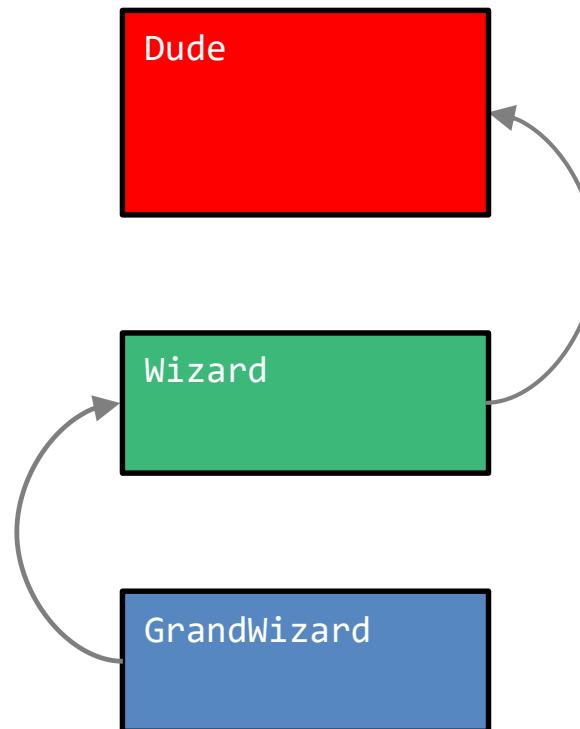
It's not there! Does Wizard have a parent?



# What Java Does

```
grandWizard1.punchFace(dude1);
```

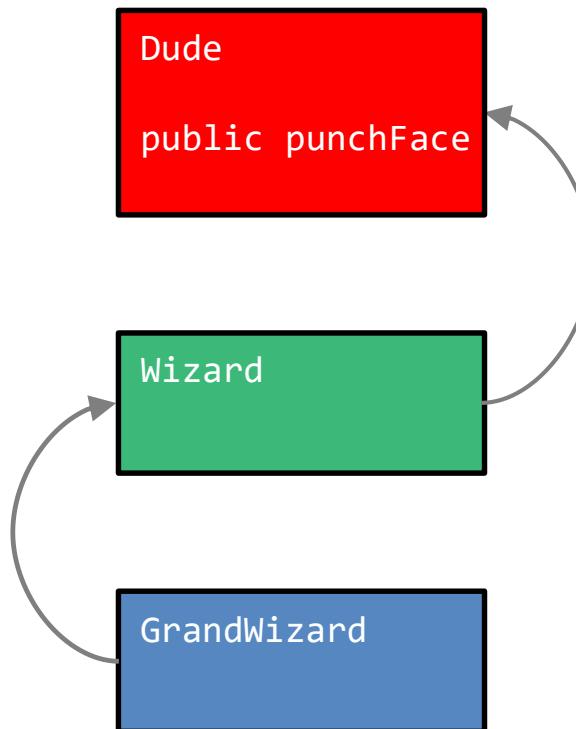
Look for punchFace in Dude class



# What Java Does

```
grandWizard1.punchFace(dude1);
```

Found it! Call punchFace



# What Java Does

```
((Dude)grandWizard1).sayName()
```

# What Java Does

```
((Dude)grandWizard1).sayName()
```

Cast to Dude tells Java to start looking in Dude



Dude

# What Java Does

```
((Dude)grandWizard1).sayName()
```

Found it! Call sayName

Dude

public sayName

# Summary

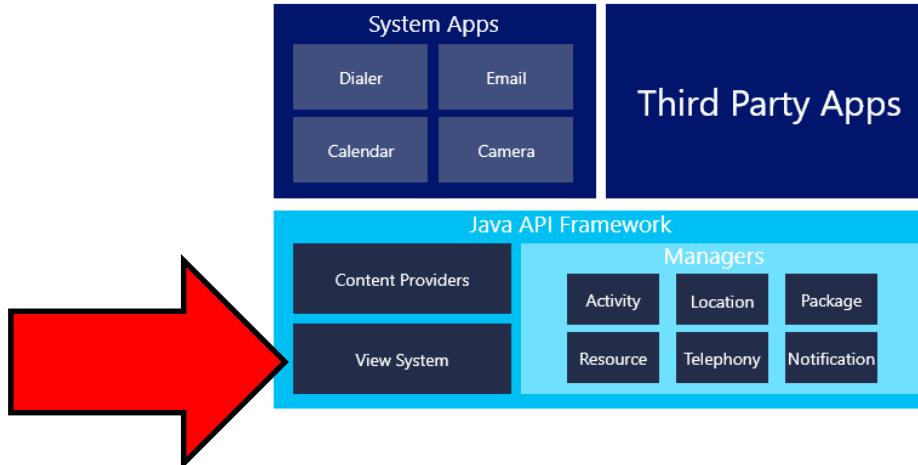
1. Class A extends B means A is a subclass of B
2. A has all the fields and methods that B has
3. A can add it's own fields and methods
4. A can only have 1 parent
5. A can replace a parent's method by re-implementing it
6. If A doesn't implement something Java searches ancestors

# Views



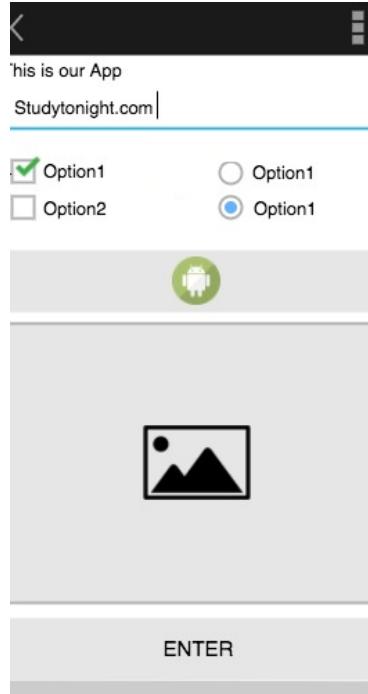
# View System

- The Java API Framework comes with a **built-in** View System



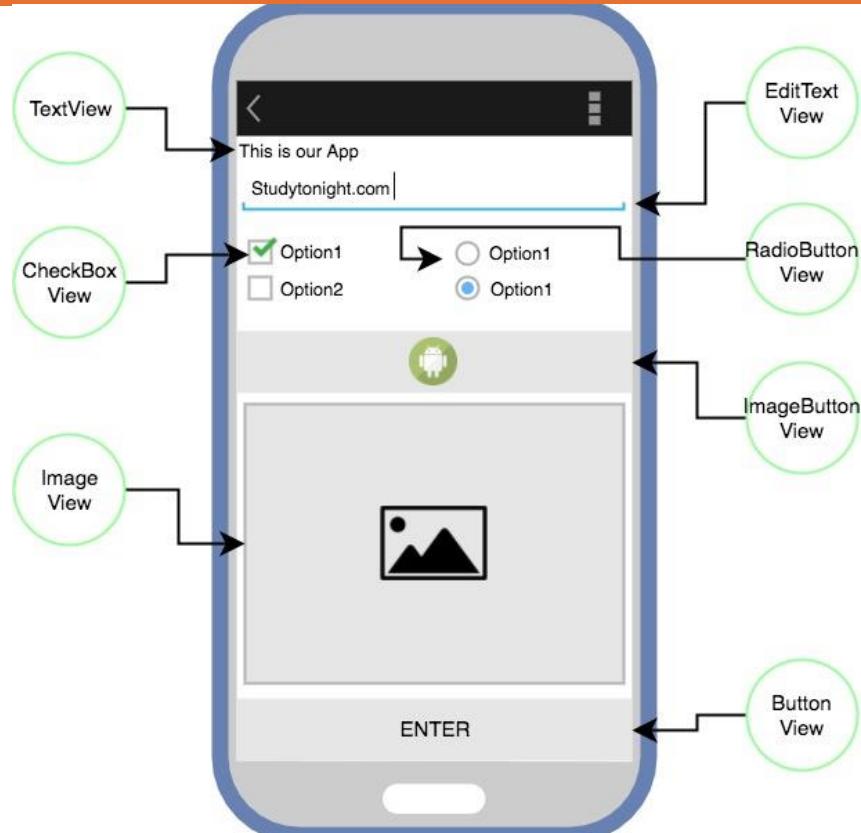
# Examples

If you look at your mobile device, every **User Interface** element that you see is a **View**.



# Examples

If you look at your mobile device, every **User Interface** element that you see is a **View**.



# What is a View?

View **subclasses** are basic User Interface building blocks

- Display text ([TextView](#) class), Edit text ([EditText](#) class)
- Buttons ([Button](#) class), [menus](#), other controls
- Scrollable ([ScrollView](#), [RecyclerView](#))
- Show images ([ImageView](#))
- Group views ([ConstraintLayout](#) and [LinearLayout](#))

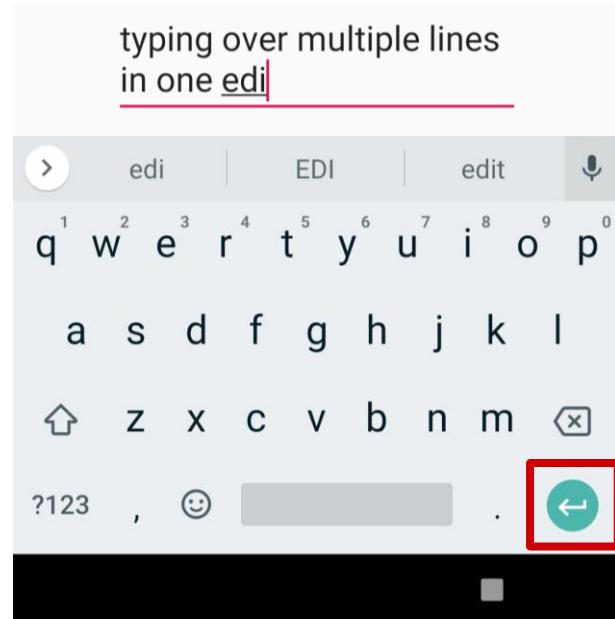
# Button

- View that responds to **tapping** (clicking) or pressing
- Usually text or visuals indicate what will happen when tapped



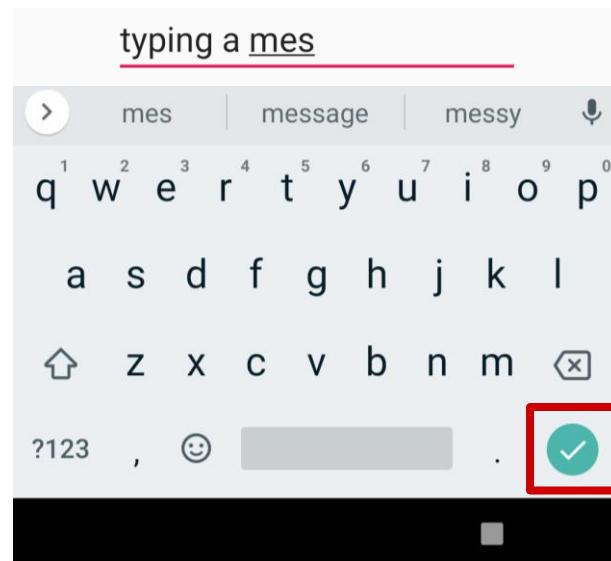
# EditText

- EditText default lines of text
- Alphanumeric Keyboard
- Suggestions appear
- Tapping **Return** key starts new line
- Set in Attributes pane of Layout Editor



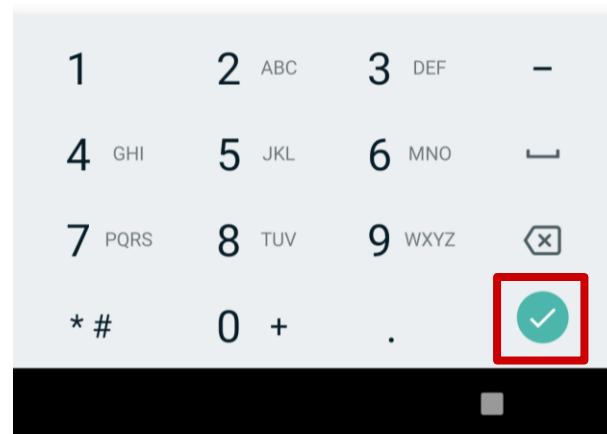
# Messages

- Single line of text
- `android:inputType = "textShortMessage"`
- Tapping Emoticons key changes keyboard to emoticons
- Tapping **Done** key advances focus to next View



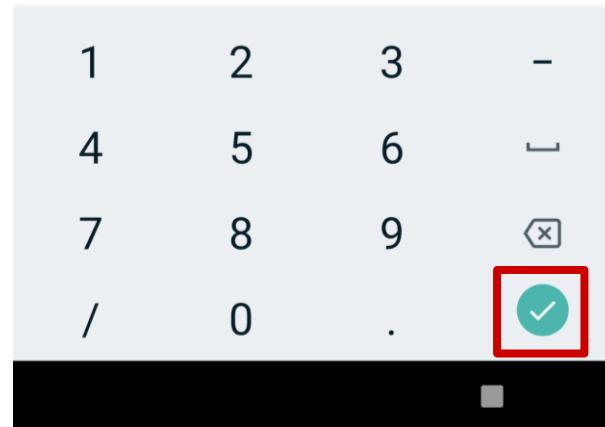
# Phone Numbers

- For phone number entry
- android:inputType = “phone”
- Numeric keypad (numbers only)
- Tapping **Done** key advances focus to next View



# Dates

- For date entry
- android:inputType = “date”
- Numeric keypad (numbers only)
- Tapping **Done** key advances focus to next View
- Note prominence of **backslash /**



# Input types

- `textCapCharacters`: Set to ALL CAPITAL LETTERS
- `textPassword`: Conceal an entered p\*\*\*\*\*d
- `number`: **Restrict** text entry to numbers
- `textEmailAddress`: Show keyboard with **@** conveniently located
- `datetime`: Show a numeric keypad with a slash and colon for entering the date and time.

# CheckBox

- User can select **any** number of choices
- Checking one box does **not** uncheck another
- Users expect checkboxes in a **vertical** list
- Commonly used with a **Submit** button

<input checked="" type="checkbox"/>	Chocolate Syrup
<input checked="" type="checkbox"/>	Sprinkles
<input type="checkbox"/>	Crushed Nuts

# RadioButton

- Put RadioButton elements in a RadioGroup in a vertical list (horizontally if labels are short)
- User can select **only one** of the choices
- Checking one **unchecks** all others
- Commonly used with a **Submit** button for the RadioGroup

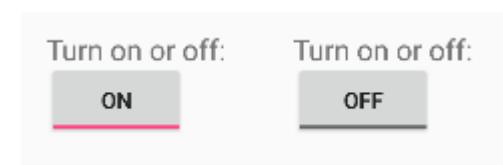
Choose a delivery method:

- Same day messenger service
- Next day ground delivery
- Pick up

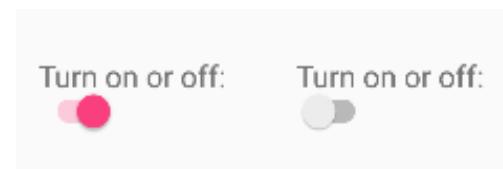
# Toggle Buttons / Switches

- User can switch between On and Off

Toggle buttons



Switches



# Custom Views

- Over 100 different types of Views available from the Android system, all **children** of the View class
- If necessary, create custom views by **subclassing** existing views or the View class itself

# View Attributes

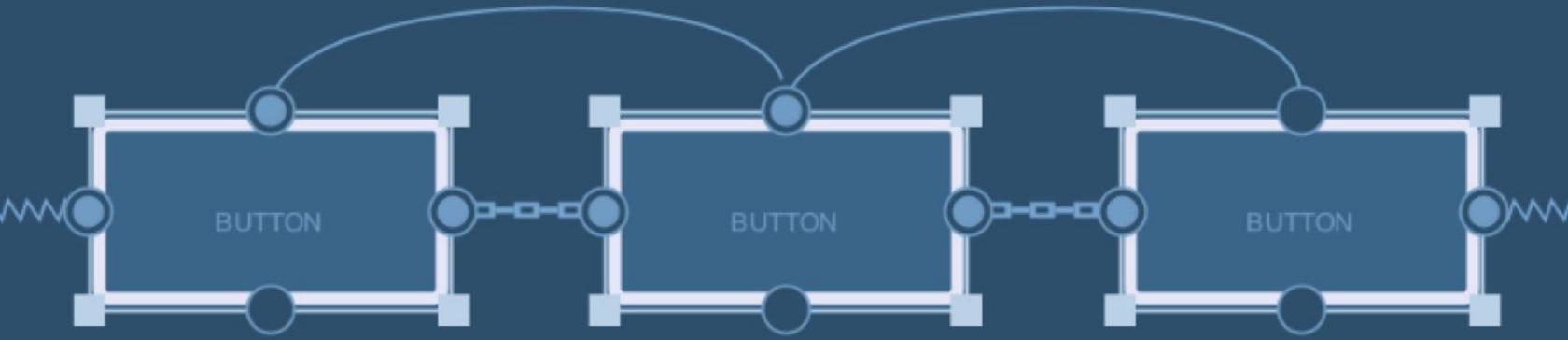
- Every View and ViewGroup object supports their own set of attributes.
- Some attributes are specific to a View object e.g. TextView supports the textSize attribute.
- Some are common to all View objects, because they are **inherited** from the root View class e.g. the **id** attribute.

# Create Views and Layouts

1. Android Studio Layout Editor: visual representation of XML
2. XML Editor
3. Java code

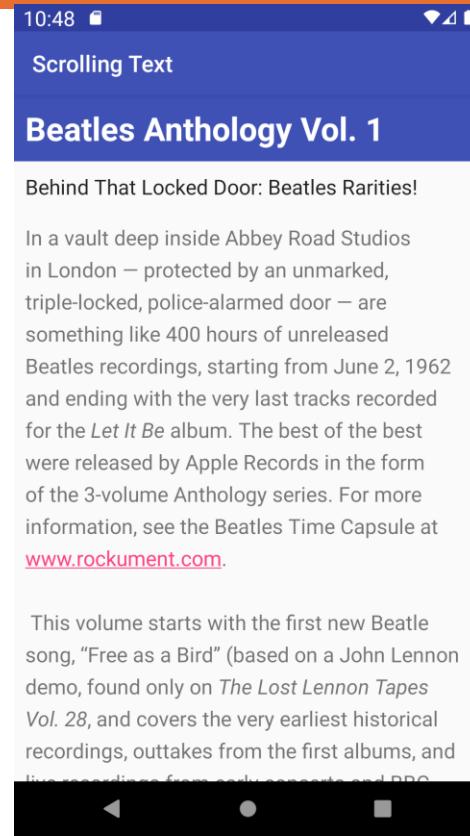


# ViewGroup and View hierarchy



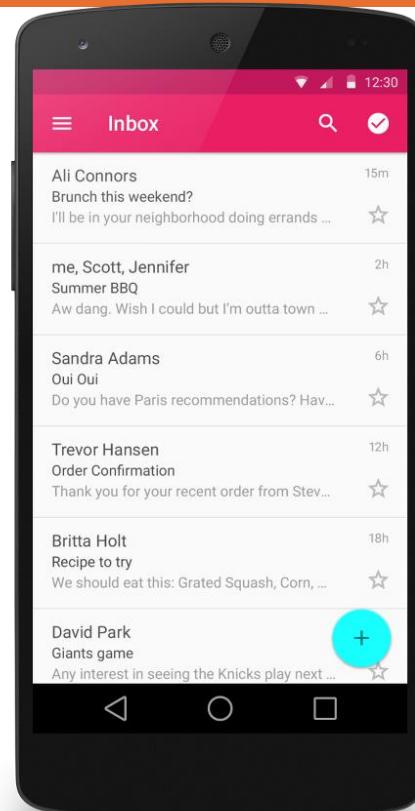
# ScrollView

- Allows View placed within it be scrolled.
- Only 1 direct child placed within it.
- To add multiple views make the direct child a view group e.g. LinearLayout.
- Supports **vertical** scrolling only.



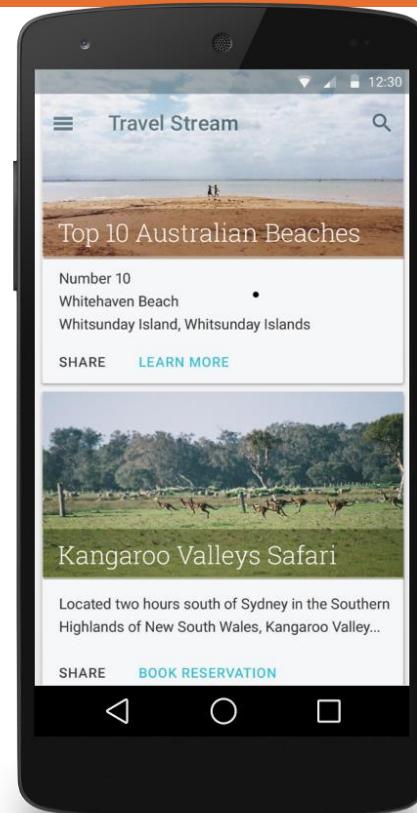
# Recycler View

- If your app needs to display a scrolling list of elements based on large data sets (or data that frequently changes).
- Replacement for older ListView.



# Recycler View

- Cards provide an easy way to contain a group of views while providing a consistent style for the container.
- Lists of these cards can then be displayed in a Recycler View.

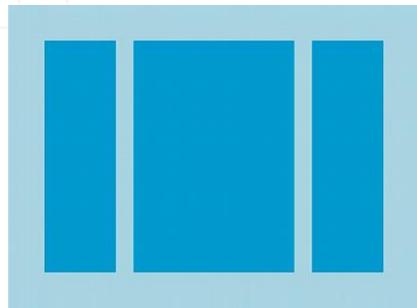


# ViewGroups for Layouts

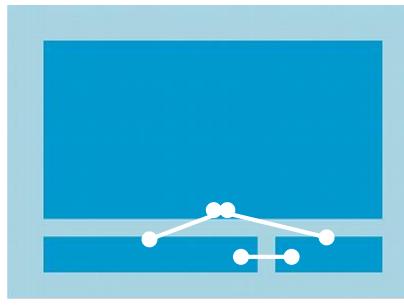
## Layouts

- Are specific types of ViewGroups (subclasses of [ViewGroup](#))
- Contain child views
- Can be in a row, column, grid, table, absolute

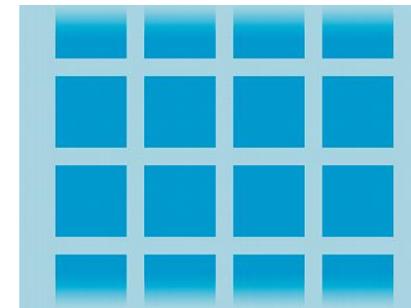
# Common Layout Classes



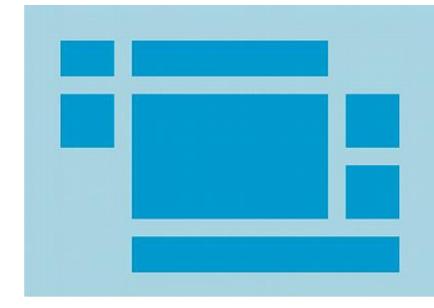
LinearLayout



ConstraintLayout



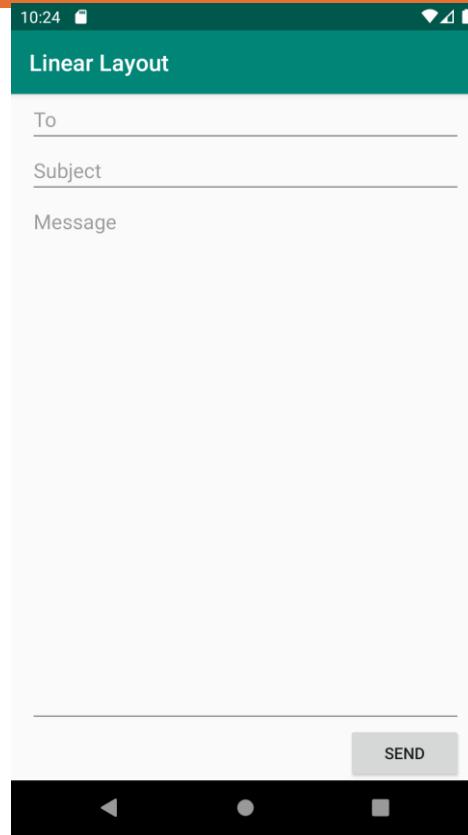
GridLayout



TableLayout

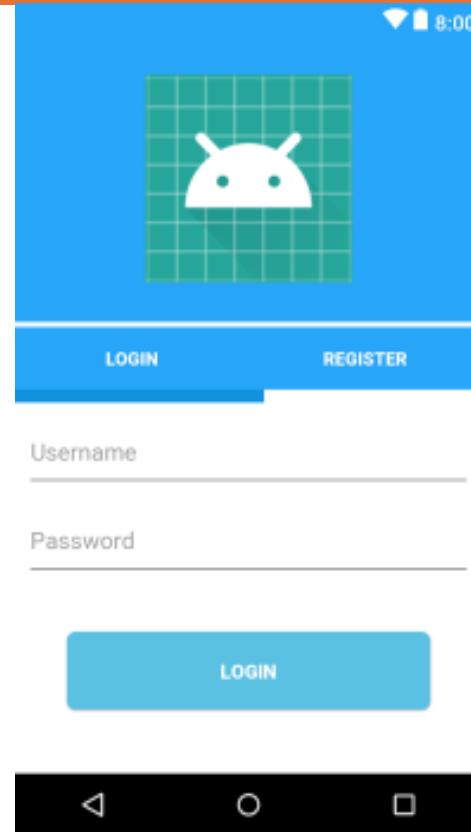
# Linear Layout

- Aligns all children in a **single** direction, vertically or horizontally.
- Specify layout direction with **orientation** attribute.
- Vertical list will only have 1 child per row.
- Respects margins between children and **gravity** (alignment) of each child.

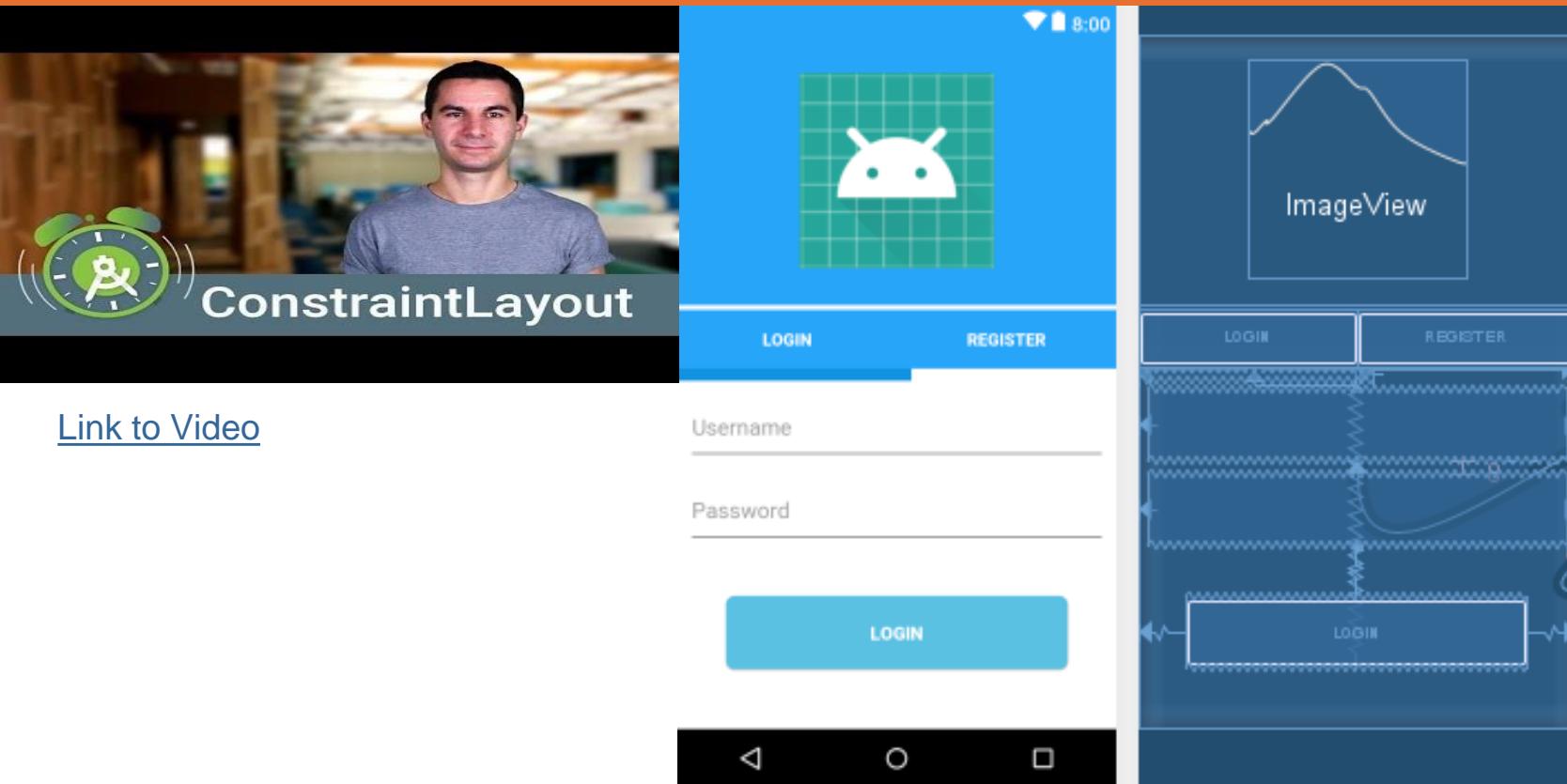


# Constraint Layout

- Create large and complex layouts with a **flat** view hierarchy (no nested view groups).
- Laid out according to relationships between **sibling** views and the parent layout.
- **Layout Editor** specially built for it.
- Drag and drop v's XML editing



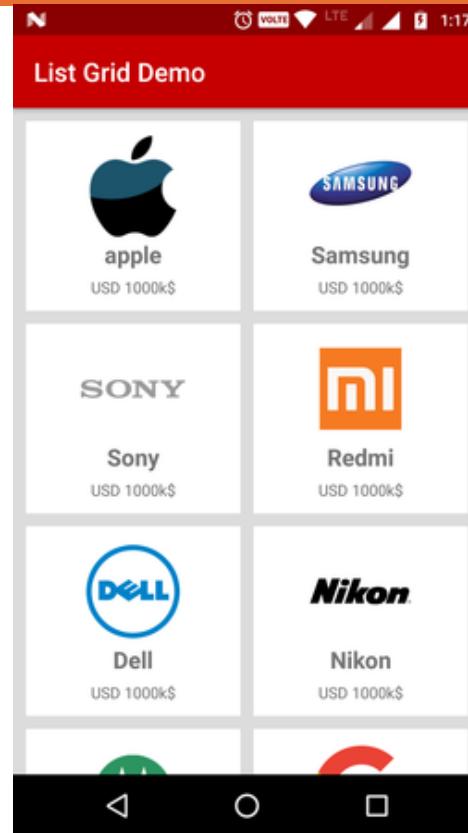
# Constraint Layout



[Link to Video](#)

# Grid Layout

- A layout that places its children in a rectangular grid.
- The grid **separates** the viewing area into cells.
- Children occupy one or more contiguous cells.



# Table Layout

- Arranges children into rows and columns.
- Consists of TableRow objects, each defining a **row** of 0 or more cells.
- A cell can hold one View object.
- Has as many **columns** as row with most cells.

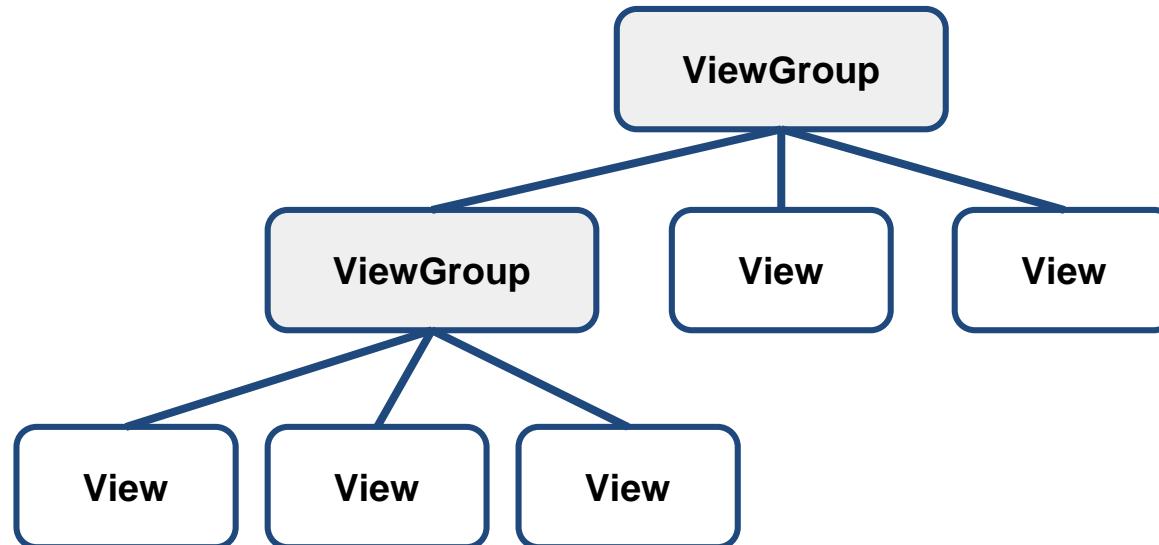


# Class and Layout Hierarchies

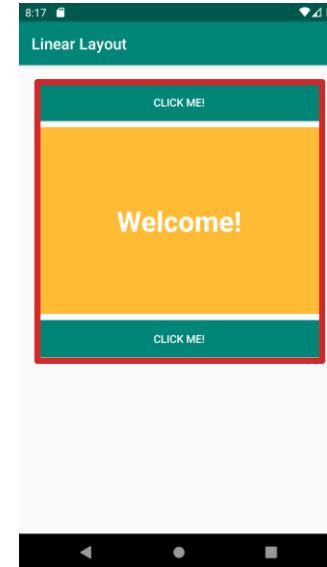
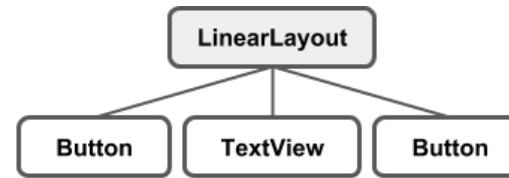
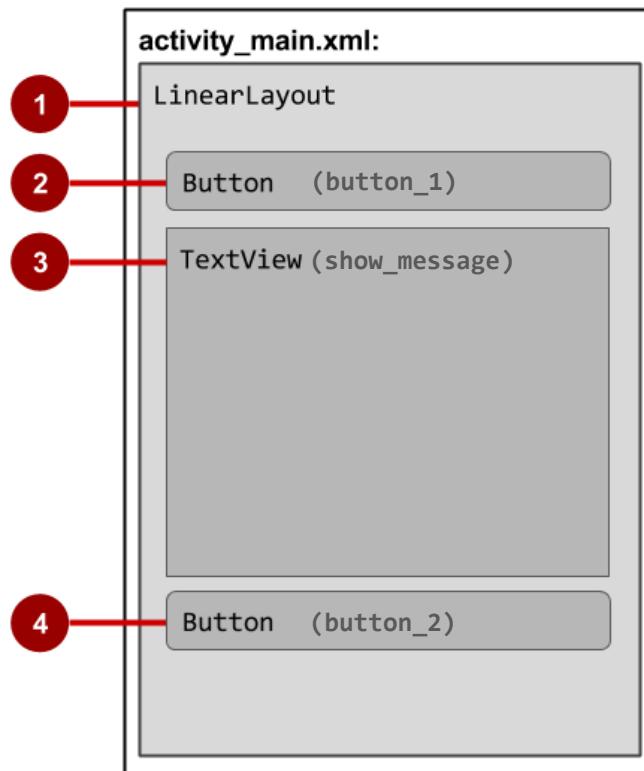
- View class-hierarchy is standard **object-oriented** class Inheritance
  - For example, Button is-a TextView is-a View is-an Object
  - Superclass-subclass relationship
- Layout hierarchy is how Views are visually arranged
  - LinearLayout can **contain** Buttons arranged in a row
  - Parent-child relationship

# Hierarchy of Viewgroups / Views

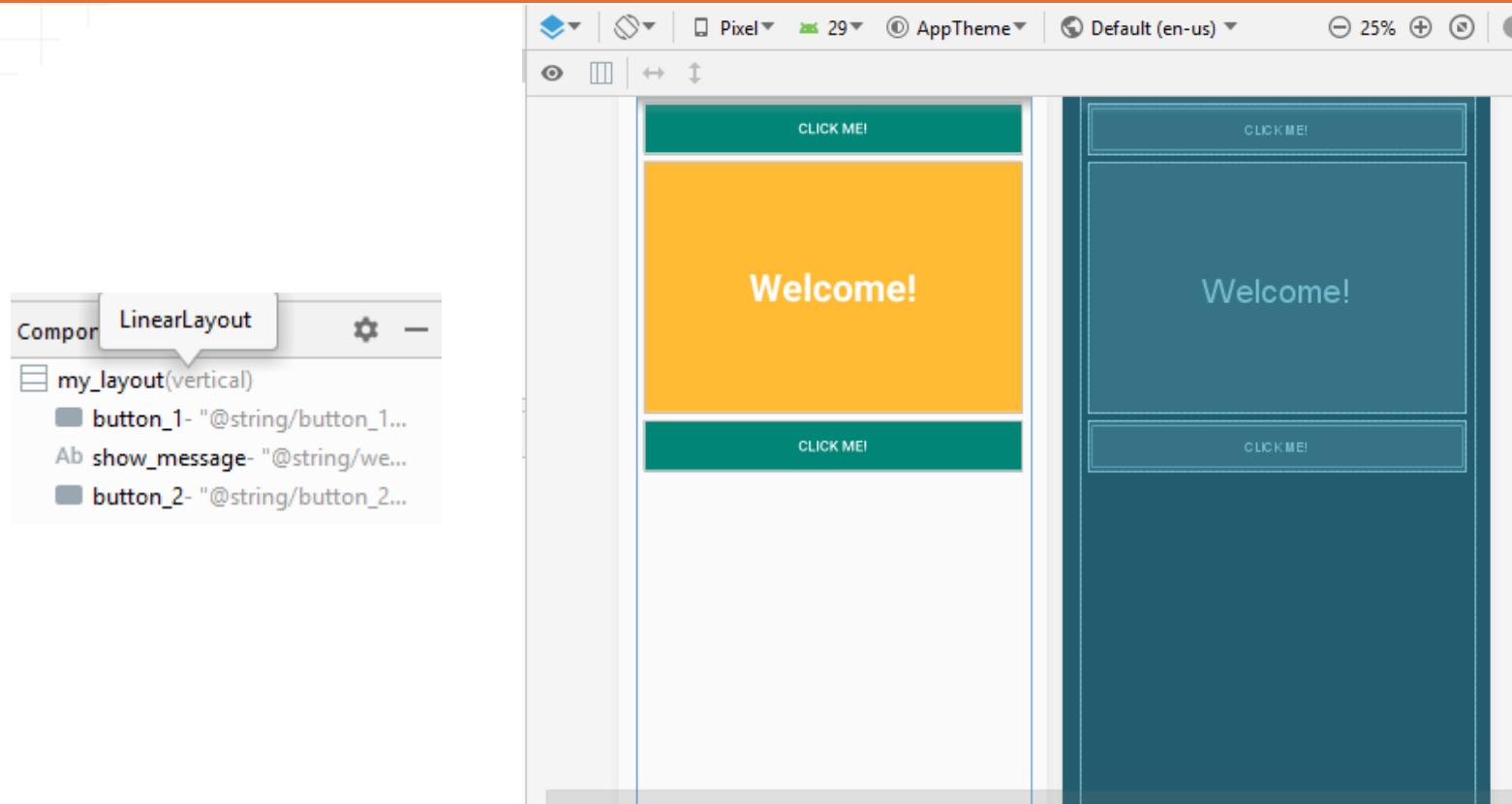
Root View is always a ViewGroup



# View hierarchy and screen layout



# In the Layout Editor



# Layout created in XML

```
<LinearLayout  
    android:orientation="vertical"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
    <Button  
        ... />  
    <TextView  
        ... />  
    <Button  
        ... />  
</LinearLayout>
```

# Layout created in Java Code

```
LinearLayout LL = new LinearLayout(this);
LL.setOrientation(LinearLayout.VERTICAL);

TextView myText = new TextView(this);
myText.setText("Display this text!");

LL.addView(myText);
setContentView(LL);
```

# Note

- Context is an interface to global information about an application environment

- Get the context:

```
Context context = getApplicationContext();
```

- An Activity is its own context:

```
TextView myText = new TextView(this);
```

# Best Practices

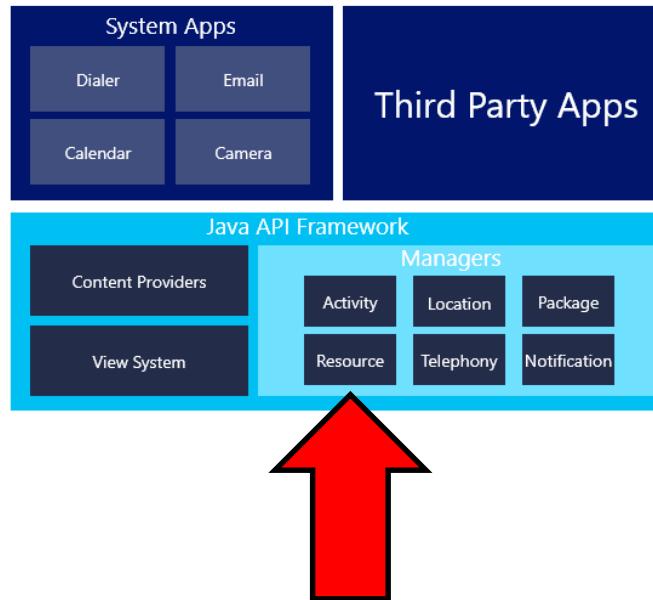
- Arrangement of View hierarchy affects app performance
- Use smallest number of simplest views possible
- Keep the hierarchy flat i.e. limit nesting of views and view groups

# Resources

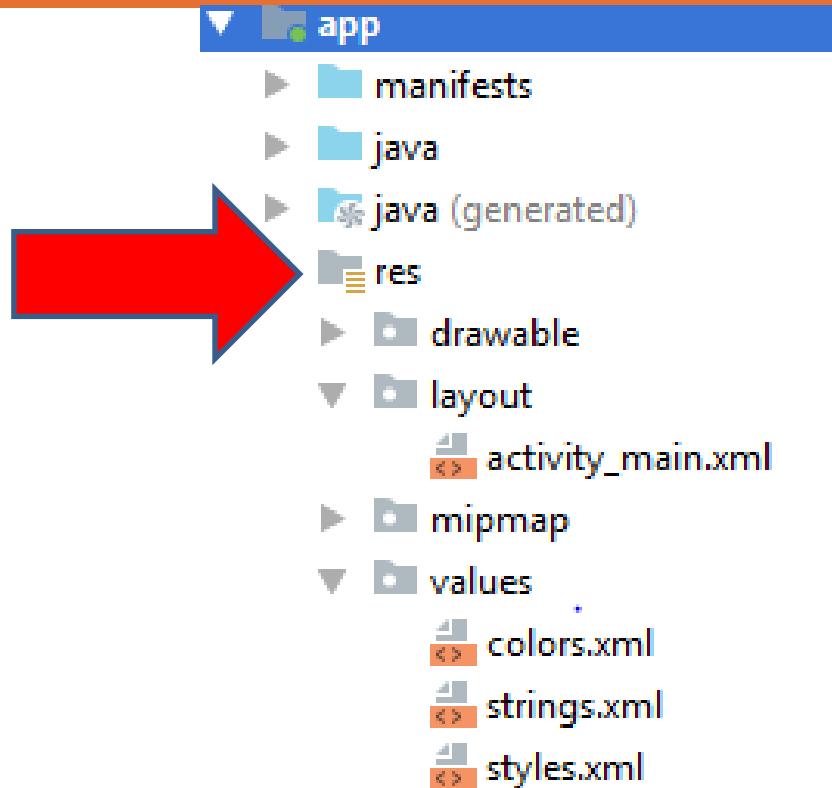
- Separate **static** data from code in your [layouts](#).
- Strings, dimensions, images, menu text, colors, styles
- Useful for localisation

# Resources

- The Java API Framework allows us easily use app Resources



# Resources



# Drawables

- Drawable—generic Android class used to represent **any kind** of graphic
- Bitmaps e.g. PNG (.png), JPG (.jpg), or GIF (.gif) format
- Vector Graphics, scale smoothly for all screen sizes e.g. defined in XML

# What is a Style?

- Collection of attributes that define the visual appearance of a View
- Reduce **duplication**
- Make code more **compact**
- Manage visual appearance of many components with one style

# Styles Reduce Clutter

```
<TextView  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:textColor="#00FF00"  
    android:typeface="monospace"  
    android:text="@string/hello" />
```

```
<TextView  
    style="@style/CodeFont"  
    android:text="@string/hello"  
/>
```

# Defined in styles.xml

**styles.xml** is in **res/values**

```
<resources>
    <style name="CodeFont">
        <item name="android:textColor">#00FF00</item>
        <item name="android:typeface">monospace</item>
    </style>
</resources>
```

# Inheritance: Parent

Define a parent style...

```
<resources>
    <style name="CodeFont">
        <item name="android:layout_width">match_parent</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:textColor">#00FF00</item>
        <item name="android:typeface">monospace</item>
    </style>
</resources>
```

# Inheritance: Define Child

Define child with Codefont as parent

```
<resources>
    <style name="RedCode" parent="@style/Codefont">
        <item name="android:textColor">#FF0000</item>
    </style>
</resources>
```

# Themes

- A Theme is a style applied to an entire activity or even the entire app
- Themes are defined in styles.xml and may be customised

```
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">  
    <!-- Customise your theme here. -->  
  
    <item name="colorPrimary">@color/colorPrimary</item>  
    <item name="colorPrimaryDark">@color/colorPrimaryDark</item>  
    <item name="colorAccent">@color/colorAccent</item>  
</style>
```

# Colour

- A color value can be defined in XML.
- The color is specified with an RGB value and alpha channel.
- You can use a color resource any place that accepts a hexadecimal color value.
- Value always begins with # character and then the Alpha-Red-Green-Blue info e.g. #RGB

# Colour Example

```
<resources>  
    <color name="colorPrimary">#008577</color>  
    <color name="colorPrimaryDark">#00574B</color>  
    <color name="colorAccent">#D81B60</color>  
</resources>
```

# Strings

Strings.xml:

```
<resources>  
    <string name="hello">Hello!</string>  
</resources>
```

Layout:

```
<TextView  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:text="@string/hello" />
```

# Refer to resources in code

- Drawable

XML: @drawable/filename

```
<ImageView  
    android:layout_height="wrap_content"  
    android:layout_width="wrap_content"  
    android:src="@drawable/myimage" />
```

Java: R.drawable.filename

```
Resources res = getResources();  
Drawable drawable = res.getDrawable(R.drawable.myimage);
```

# Refer to resources in code

- Layout:

```
setContentView(R.layout.activity_main);
```

- View:

```
rv = (RecyclerView) findViewById(R.id.recyclerview);
```

- String:

Java: R.string.title

XML: android:text="@string/title"

# Refer to resources in code

1. Get a EditText object for the EditText View

```
EditText simpleEditText =  
    findViewById(R.id.edit_simple);
```

2. Retrieve the CharSequence and convert it to a string

```
String strValue =  
    simpleEditText.getText().toString();
```

# Measurements

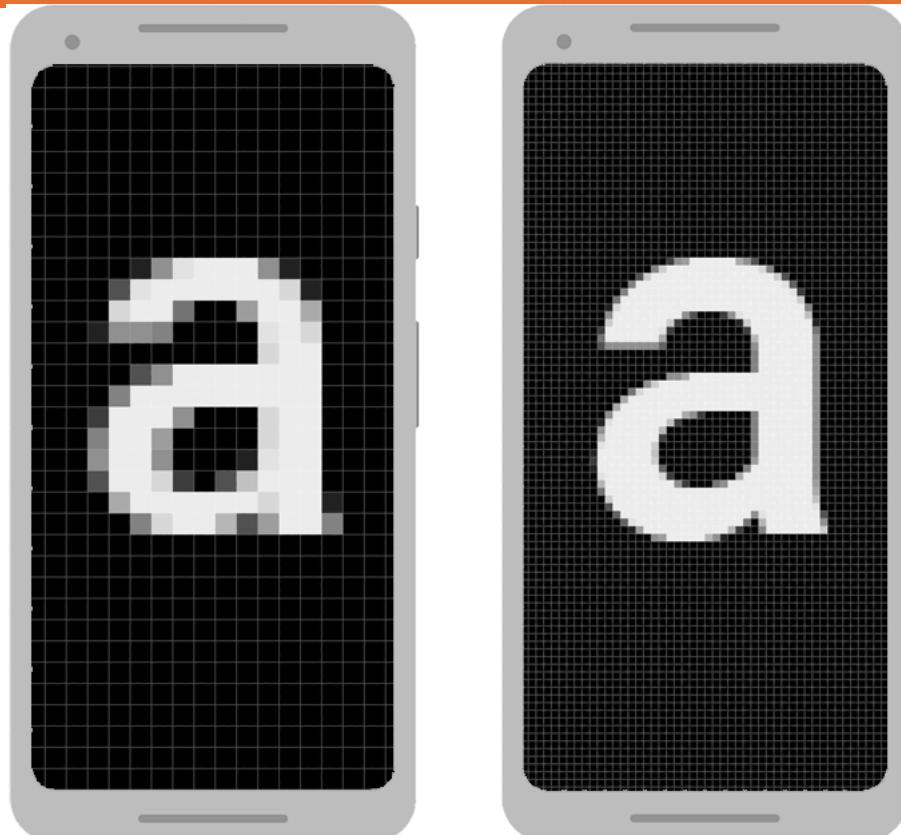


# Measurements

- Devices have different screen sizes (handsets, tablets, TVs...)
- Different pixel sizes e.g. 160 or 480 per square inch.
- If you don't consider this, system might:
  - Scale your images (resulting in blurry images).
  - Images might appear at the completely wrong size.

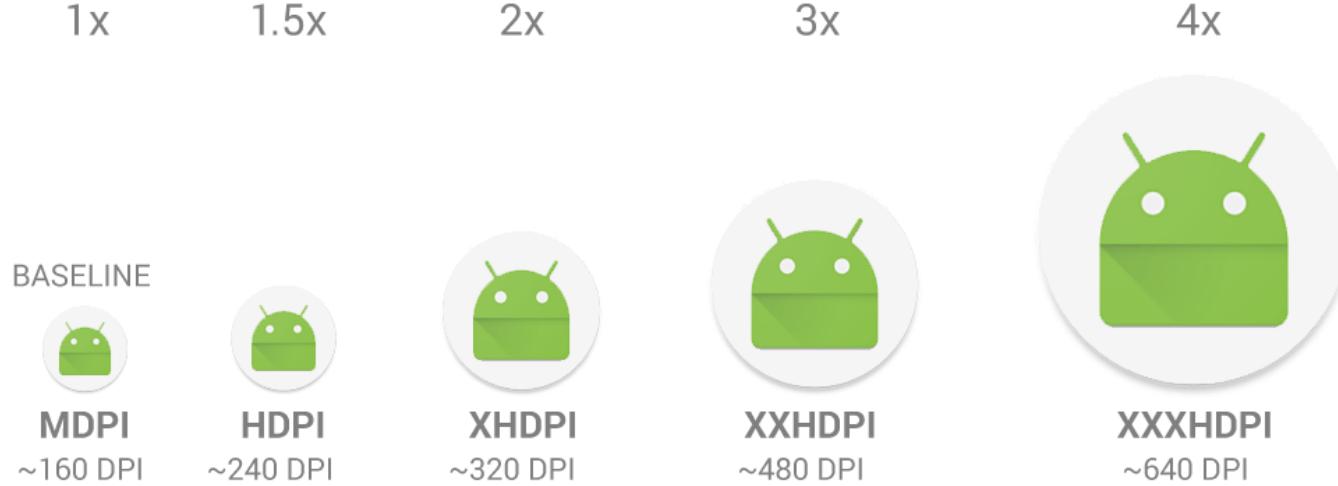
# Measurements

- A View defined as "100px" wide will appear much larger on the left device.



# Measurements

- For good **graphics quality** across devices, provide **multiple** versions of each bitmap in your app.



# dp

- 1 dp is a virtual pixel unit, roughly equal to 1 pixel on a medium-density screen (160dpi; the "baseline" density).
- Android translates dps to the number of real pixels for each other density using this formula:

$$px = dp * (dpi / 160)$$

# sp

- When defining text sizes you should instead use scalable pixels (sp) as your units (but never use sp for layout sizes).
- The sp unit is the same size as dp, by default, but it resizes based on the user's preferred text size.

# Guidelines

- Density-independent Pixels (dp): for Views
- Scalable Pixels (sp): for text

Don't use device-dependent or density-dependent units:

- Actual Pixels (px)
- Actual Measurement (in, mm)
- Points - typography 1/72 inch (pt)

# Next Week



다음주

# Mobile Software Development

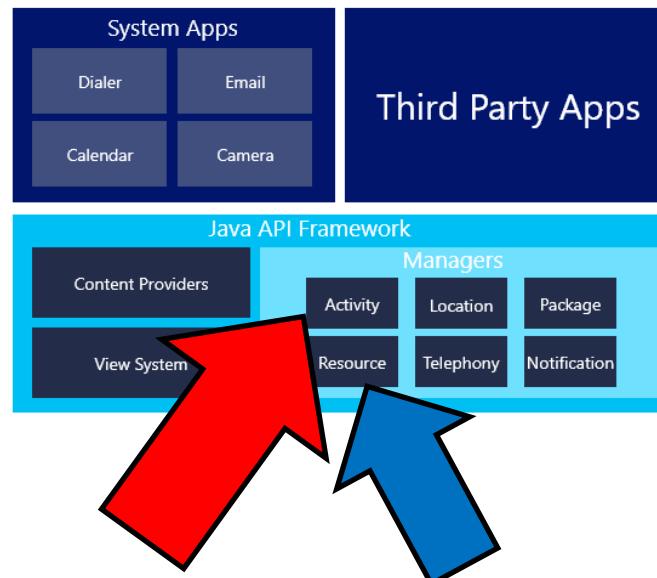
**CMPU 3036 DT211C/3, DT282/3**

# Today's Lecture

- Activities
  - Defining Activities
  - Starting Activities with an **Intent**
  - Passing **data** between Activities
  - Navigating between Activities
- **Handling Clicks**

# Framework

- The Java API Framework allows us to work with **Activities** and Resources.



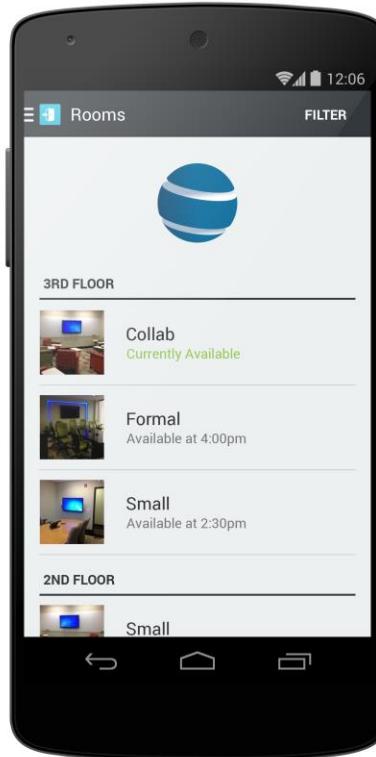
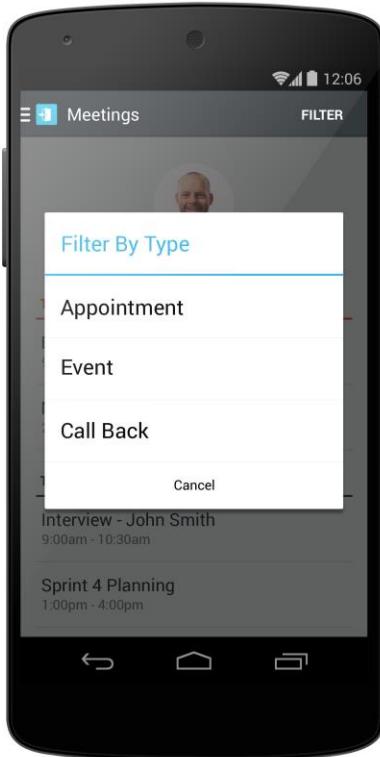
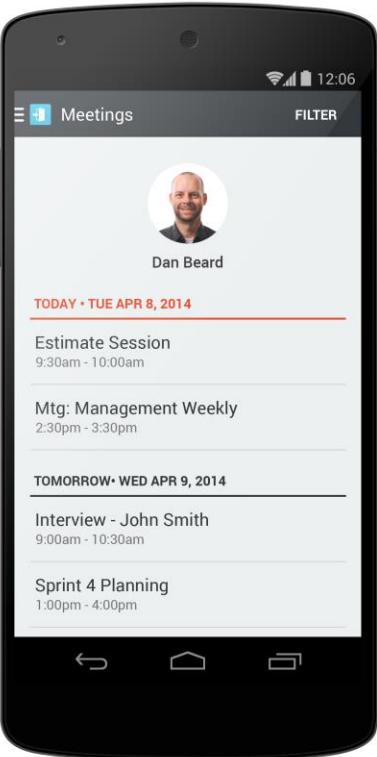
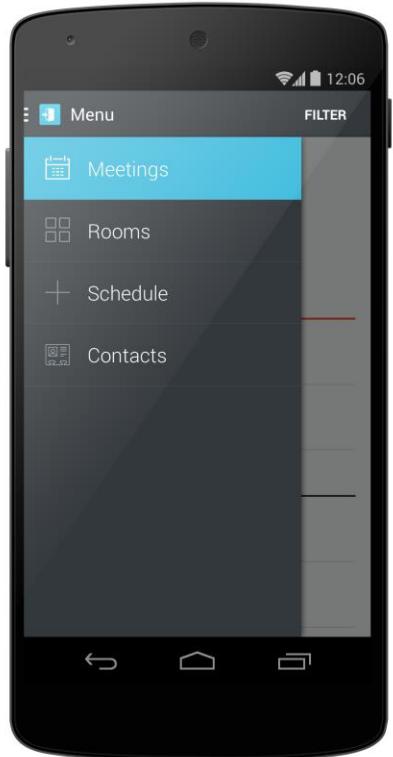
# What is an Activity?

- An Activity is an App **Component**.
- Represents one window, one hierarchy of **Views**.
- Usually fills the screen, but can be embedded in other Activity or appear as floating window.
- **Java** class, usually one Activity in one .java file.

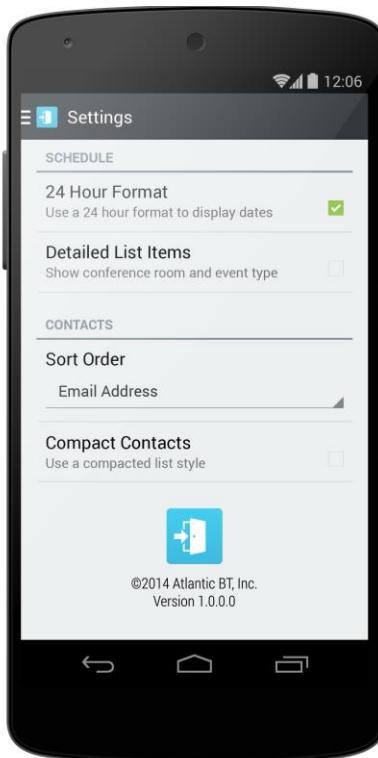
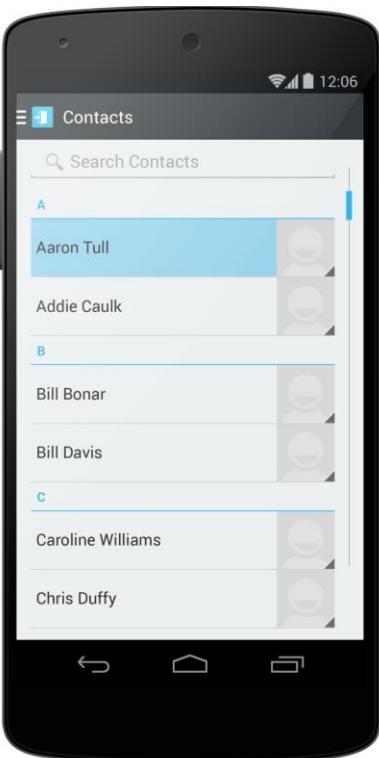
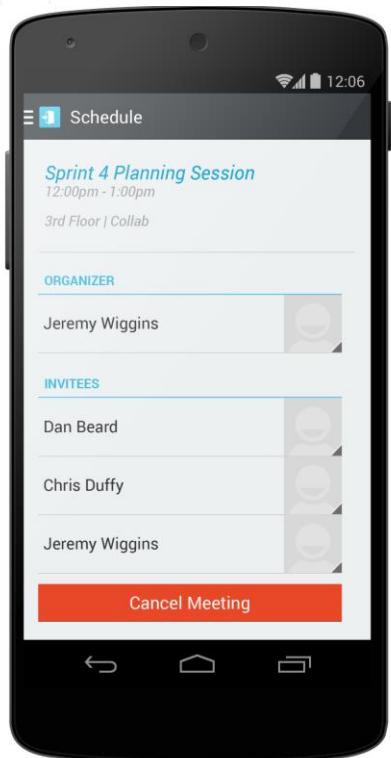
# What does an Activity do?

- Represents a User activity e.g. order groceries, send email or get directions.
- Handles user interactions e.g. button clicks, text entry etc..
- Can start **other** activities in the same or other apps.
- Has a Lifecycle - created, started, runs, paused, resumed, stopped, and destroyed.

# Example



# Example



# Apps and Activities

- Activities are **loosely** tied together to make up an App.
- First Activity user sees is usually called **Main Activity**.
- Activities can be organised in parent-child relationships in the Android manifest to aid navigation.

# Layouts and Activities

- An Activity usually has a UI **Layout**.
- Layout is usually defined in one or more XML files.
- Activity **inflates** layout as part of being created.

# Implementing Activities

# Implement new Activity

1. Define Layout in XML
2. Define Activity Java class
  - extends AppCompatActivity
3. Connect Activity with Layout
  - Set content view in onCreate()
4. Declare Activity in the Android manifest

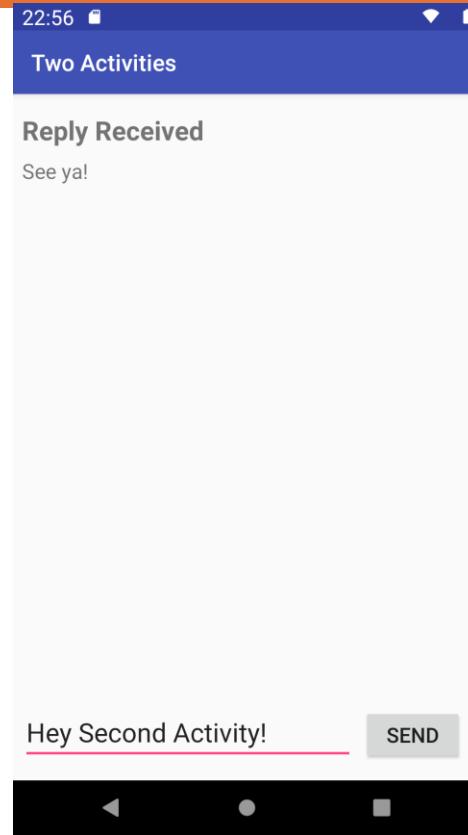
# Layout in XML

activity\_main.xml

```
<android.support.constraint.ConstraintLayout>

    <TextView
        />
    <TextView
        />
    <Button
        />
    <EditText
        />

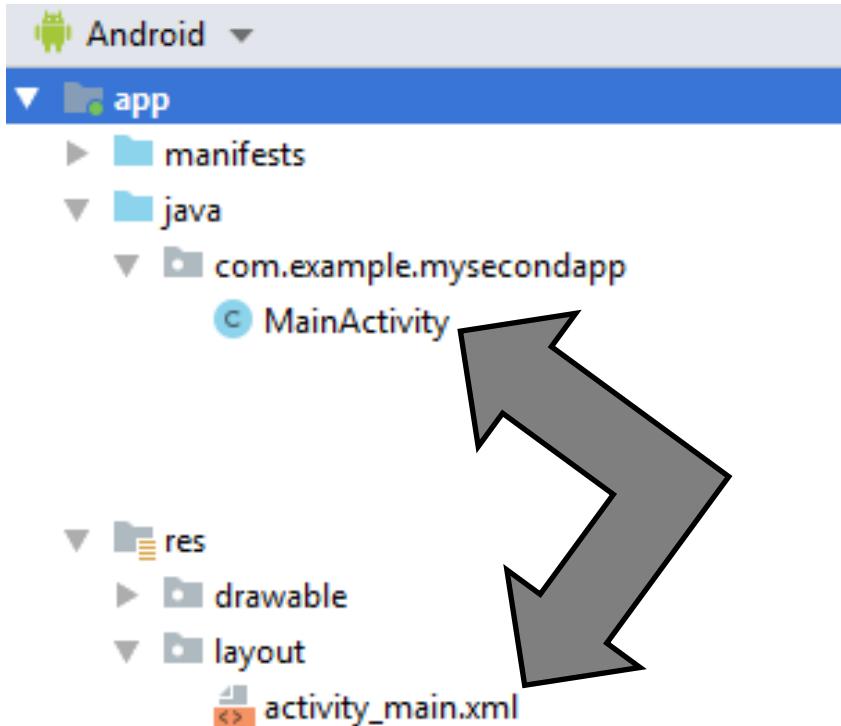
</android.support.constraint.ConstraintLayout>
```



# Activity Java Class

```
public class MainActivity extends AppCompatActivity {  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
  
    }  
}
```

# Connect Activity to XML Layout



# Connect Activity to XML Layout

```
public class MainActivity extends AppCompatActivity {  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
  
        setContentView(R.layout.activity_main);  
    }  
}
```

Resource

Layout

Name of XML Layout File

# Declare Activity in Manifest

```
<activity android:name=".MainActivity">  
  
</activity>
```

# Declare Activity in Manifest

MainActivity needs to include an **Intent Filter** to start from launcher

```
<activity android:name=".MainActivity">

    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>

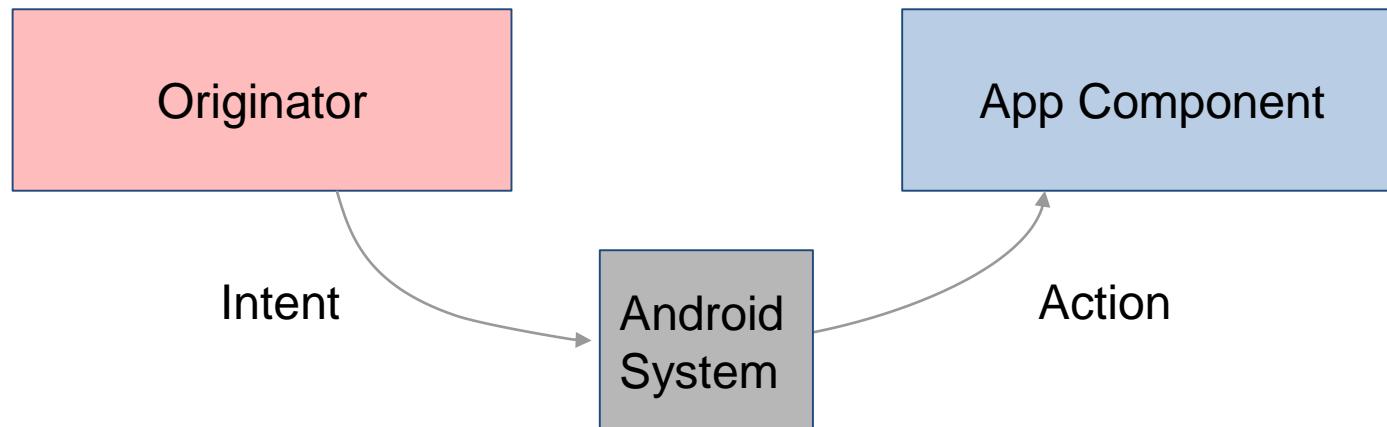
</activity>
```

# Intents

# What is an Intent?

An Intent is a **description** of an operation to be performed.

It is an object used to request an action from another app component via the Android System.



# What can Intents do?

- Start an **Activity**

A button click starts a new Activity for text entry.

Clicking Share opens an app that allows you to post a photo.

- Start a **Service**

Initiate downloading a file in the background.

- Deliver **Broadcast**

The system informs everybody that the phone is now charging.

# Starting Activities

# Types of Intent

## Explicit

- Starts a **specific** Activity

Main Activity starts the ViewShoppingCart Activity

## Implicit

- Asks system **to find** an Activity that can handle this request

Find an open store that sells green tea

Clicking Share opens a chooser with a list of apps

# Explicit Way

If you know the name of the Activity, use an Explicit Intent

1. **Create** an Intent
2. Use the Intent **to start** the Activity

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

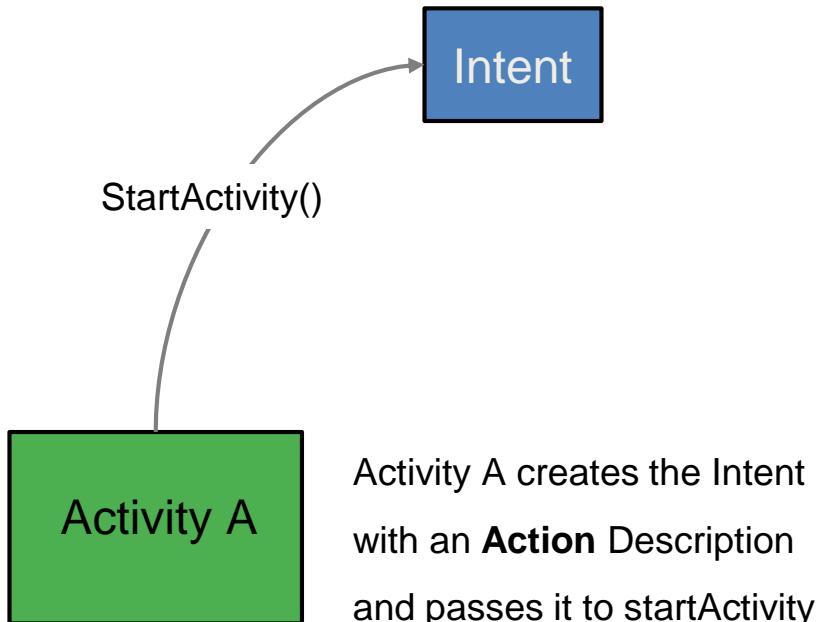
# Implicit Way

To ask Android to find an Activity to handle your request, use an implicit Intent

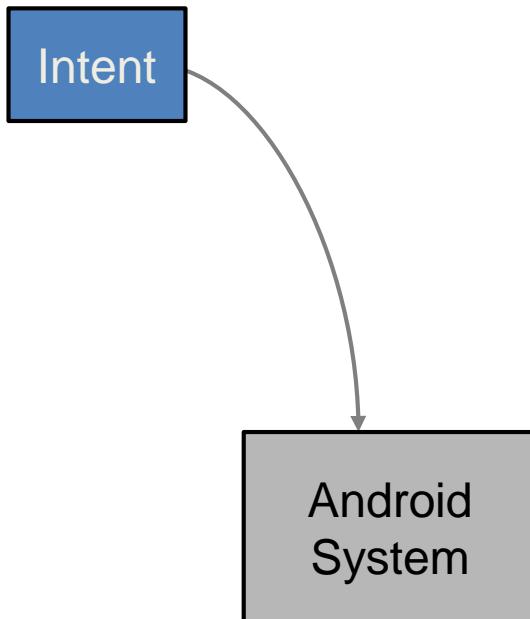
1. **Create** an Intent
2. Use the Intent **to start** the Activity

```
Intent intent = new Intent(action, uri);
startActivity(intent);
```

# How it Works



# How it Works



The Android System **searches** all Apps for an Intent Filter that **matches** the Intent.

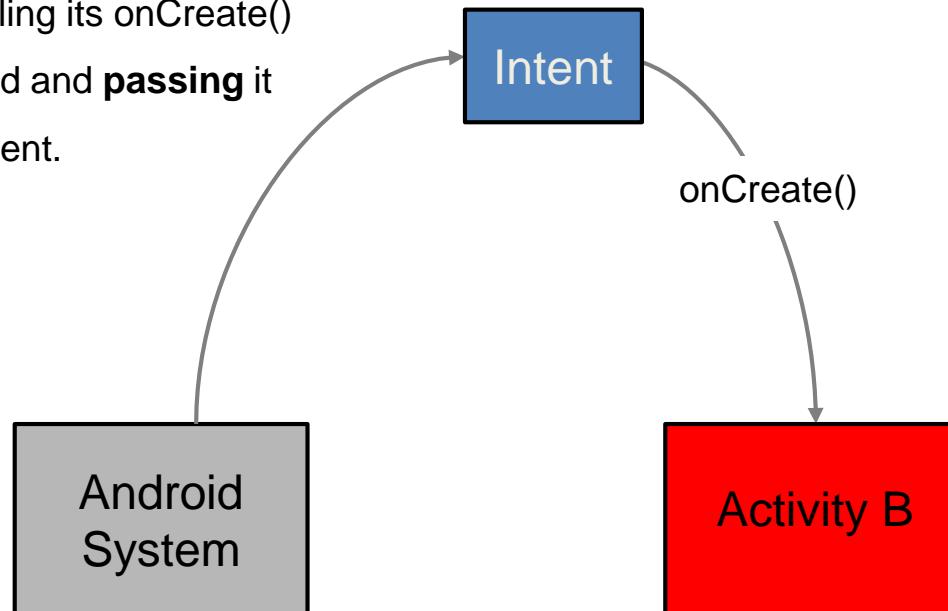
# How it Works

When a **match** is found,  
the System starts this  
activity (Activity B)

Android  
System

# How it Works

By calling its `onCreate()` method and **passing** it the Intent.



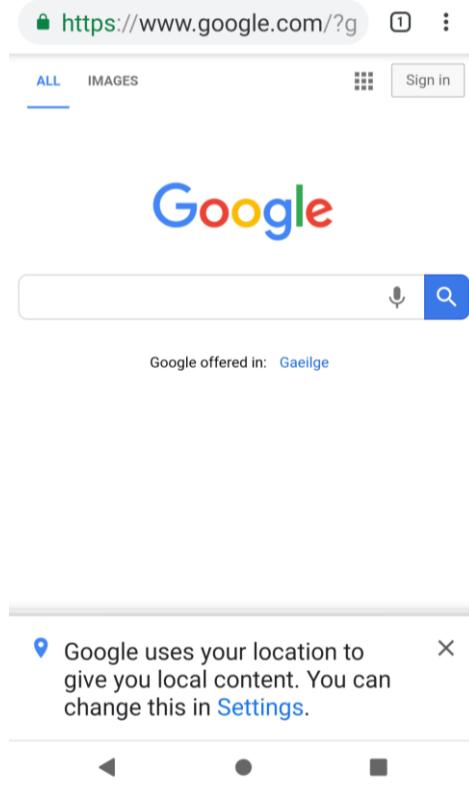
# Examples

## Show a Web Page

```
Uri uri = Uri.parse("http://www.google.com");
Intent it = new Intent(Intent.ACTION_VIEW, uri);
startActivity(it);
```

# Examples

Show a Web Page



# Examples

## Dial a Phone Number

```
Uri uri = Uri.parse("tel:8005551234");
Intent it = new Intent(Intent.ACTION_DIAL, uri);
startActivity(it);
```

# How Activities Run

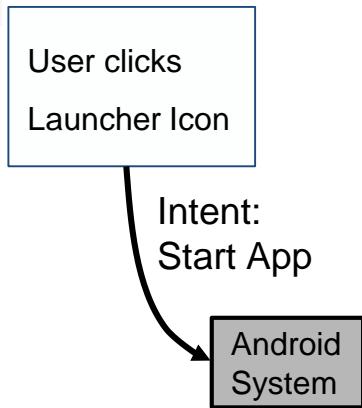
- All Activity **objects** are managed by the Android System
- Started by an **Intent**, a message to the system to run an Activity

# How Activities Run

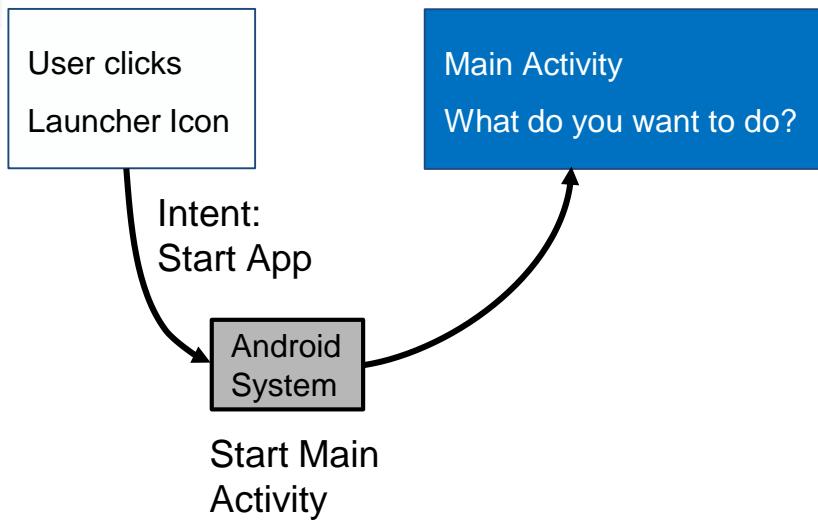
User clicks

Launcher Icon

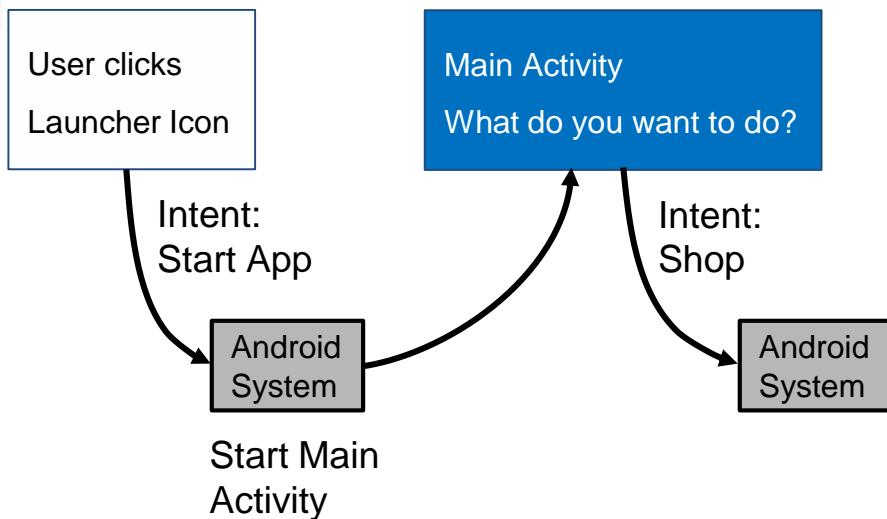
# How Activities Run



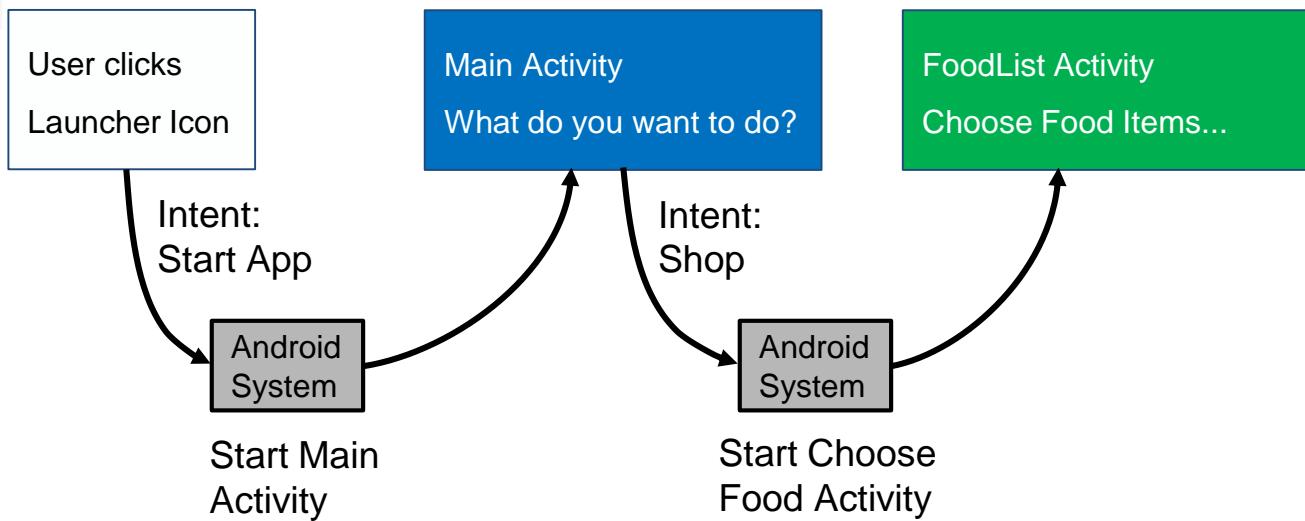
# How Activities Run



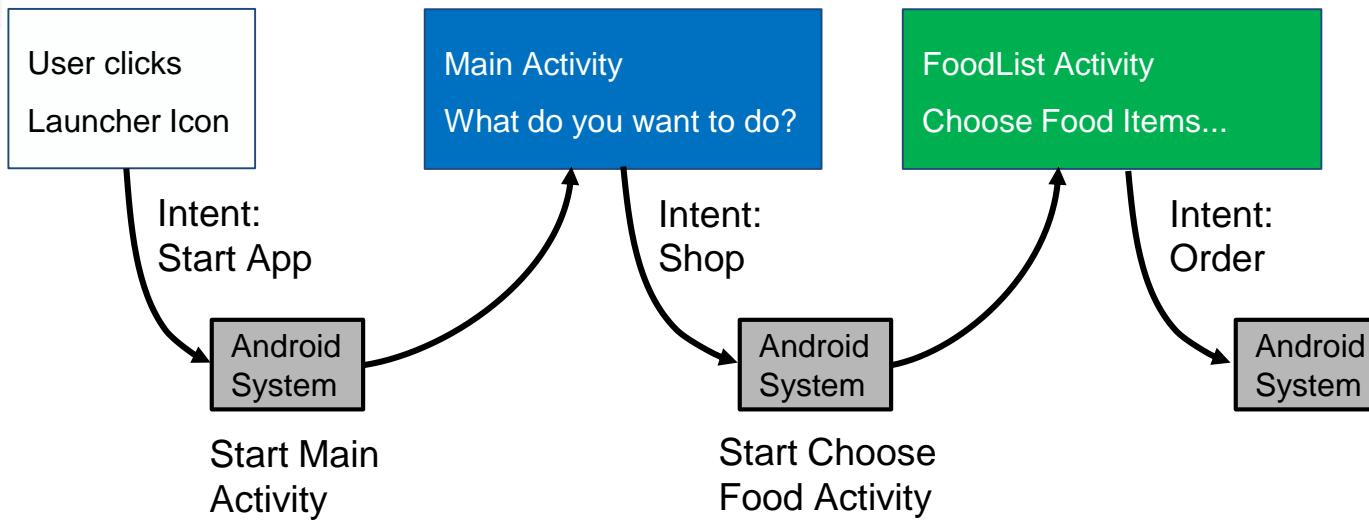
# How Activities Run



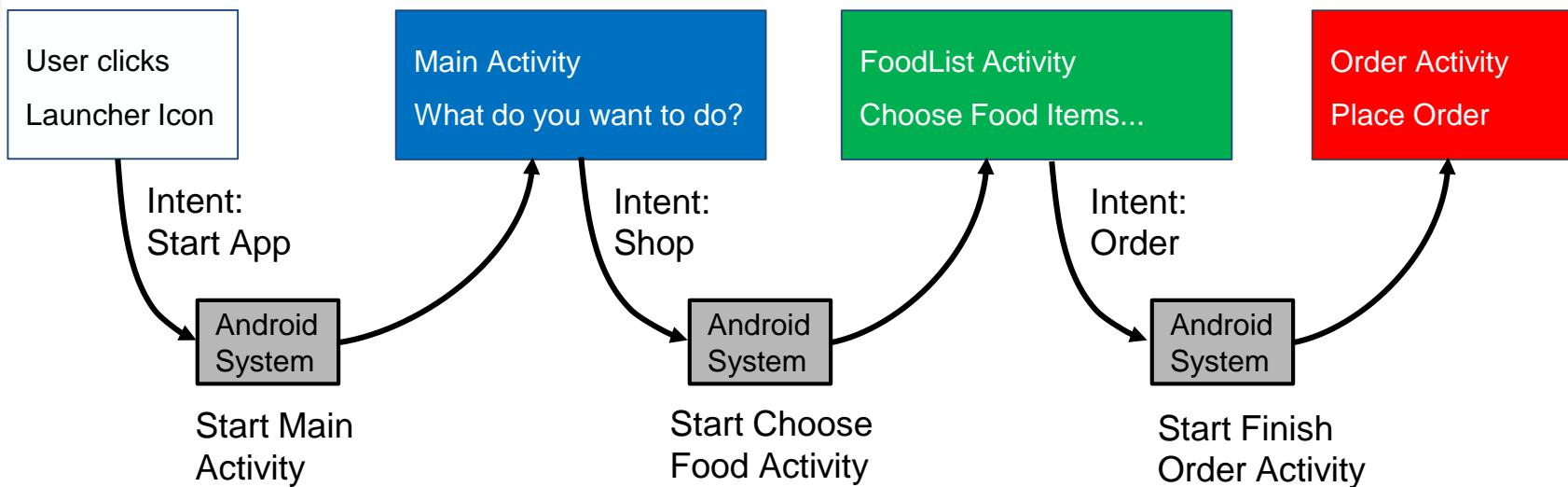
# How Activities Run



# How Activities Run



# How Activities Run



# Sending and Receiving Data

# Two Ways

1. **Data** - one piece of information whose data location can be represented by an URI
2. **Extras** - one or more pieces of information as a collection of key-value pairs in a [Bundle](#)

# How it Works

## In Activity A (**sender**)

- Create the Intent object
- Put data or extras into that Intent
- Start the new Activity with `startActivity()`

# How it Works

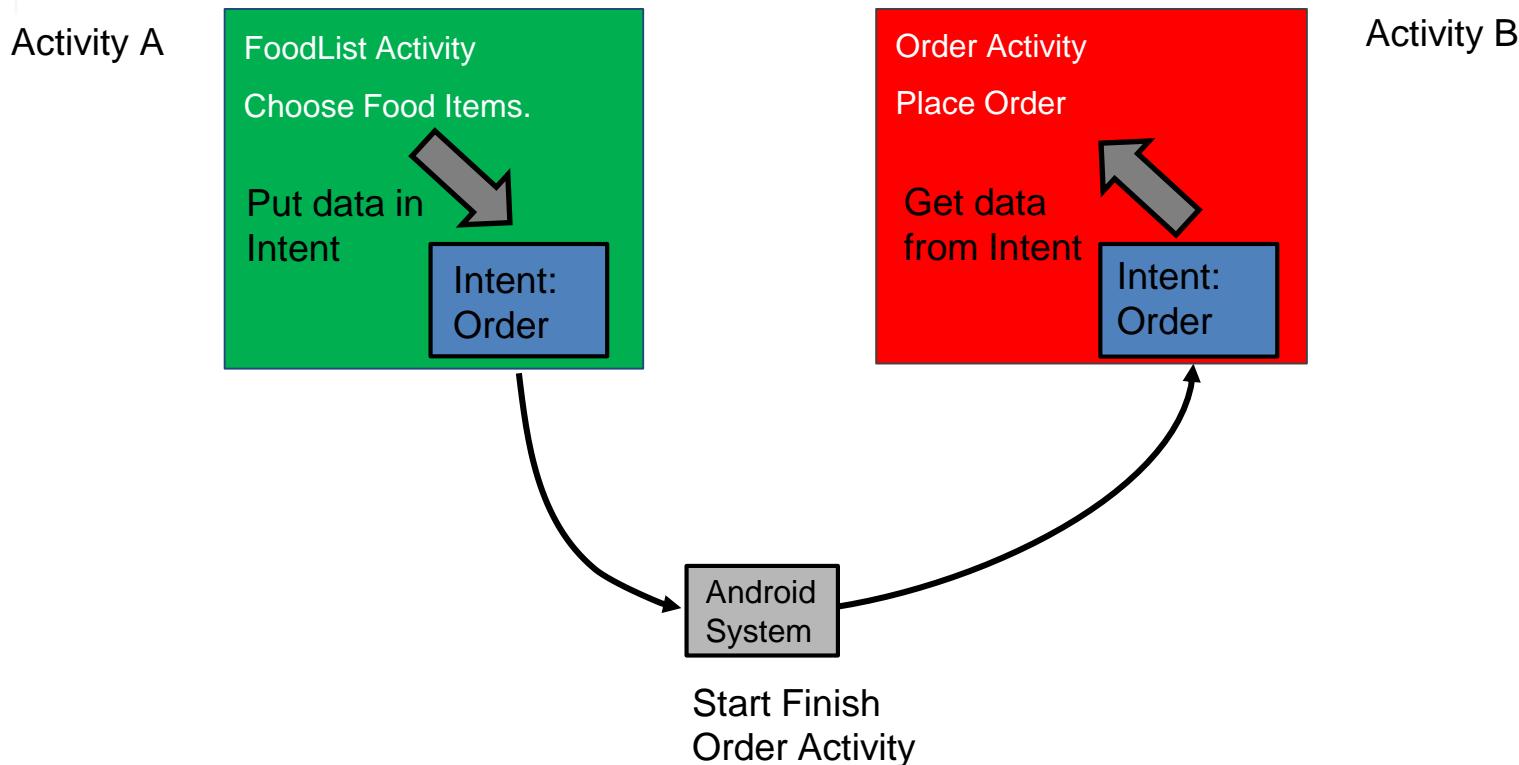
## In Activity A (**sender**)

- Create the Intent object
- Put data or extras into that Intent
- Start the new Activity with `startActivity()`

## In Activity B (**receiver**)

- Get the Intent object that Activity was started with
- Retrieve the data or extras from the Intent object

# How it Works



# URI Examples

```
// A web page URL  
intent.setData(Uri.parse("http://www.google.com"));
```

```
// a Sample file URI  
intent.setData(Uri.fromFile(new File("/sdcard/sample.jpg")));
```

See [documentation](#) for more Info

# Bundle Examples

```
putExtra(String name, int value)  
intent.putExtra("level", 406);
```

```
putExtra(String name, String[] value)  
String[] foodList = {"Rice", "Beans", "Fruit"};  
intent.putExtra("food", foodList);
```

```
// if lots of data, create bundle first and pass  
putExtras(bundle);
```

# Starting Activity

```
public static final String EXTRA_MESSAGE_KEY =  
    "com.example.android.twoactivities.extra.MESSAGE";
```

```
Intent intent = new Intent(this, SecondActivity.class);  
String message = "Hello Activity!";  
intent.putExtra(EXTRA_MESSAGE_KEY, message);
```

```
startActivity(intent);
```

# On Receiving End

```
getData();  
Uri locationUri = intent.getData();
```

```
int getIntExtra (String name, int defaultValue)  
int level = intent.getIntExtra("level", 0);
```

```
//Get all the data at once as a bundle.  
Bundle bundle = intent.getExtras();
```

# Return Data to Activity

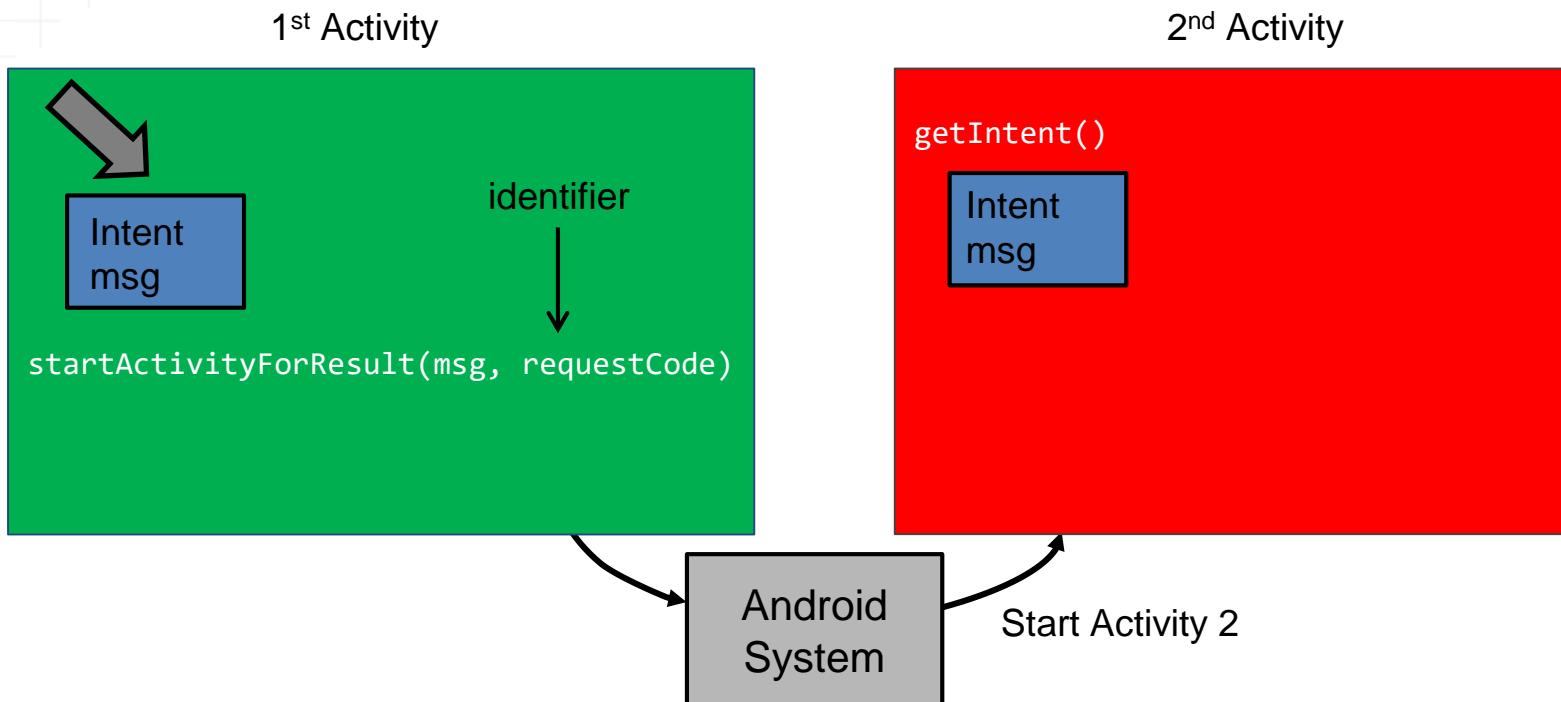
- Use `startActivityForResult()` to start the 2<sup>nd</sup> Activity
- To **return** data from the 2<sup>nd</sup> Activity:
  1. Create a new **Intent**
  2. Put response data in it with `putExtra()`
  3. Set the result to `RESULT_OK` or `RESULT_CANCELED`
  4. Call `finish()` to close Activity
- Implement `onActivityResult()` in 1<sup>st</sup> Activity

# Start Activity for Result

```
startActivityForResult(intent, requestCode);
```

- Starts Activity (intent), assigns it **identifier** (requestCode)
- Returns **data** via Intent extras
- When done, pop stack, return to previous Activity, and execute onActivityResult() **callback** to process returned data
- Use requestCode to identify **which** Activity has "returned"

# How it Works



# How it Works

2<sup>nd</sup> Activity

putExtra()



Intent  
reply

setResult(RESULT\_OK, reply);

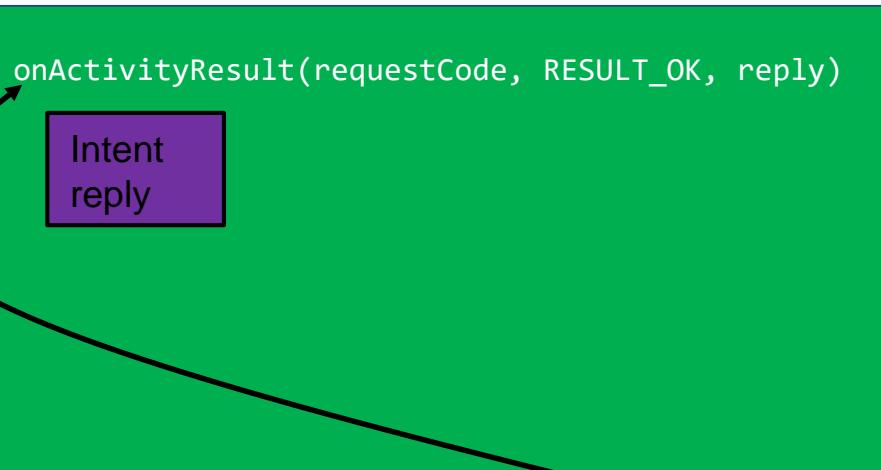
finish();

# "HOLLYWOOD PRINCIPLE":



# DON'T CALL US, WE'LL CALL YOU

1<sup>st</sup> Activity



System calls **Callback**  
method!

Android  
System

# Example Code

```
public static final int CHOOSE_FOOD_REQUEST = 1;

Intent intent = new Intent(this, ChooseFoodItemsActivity.class);
startActivityForResult(intent, CHOOSE_FOOD_REQUEST);
```

# Example Code

```
Intent replyIntent = new Intent();
replyIntent.putExtra(EXTRA_REPLY, reply);

setResult(RESULT_OK, replyIntent);

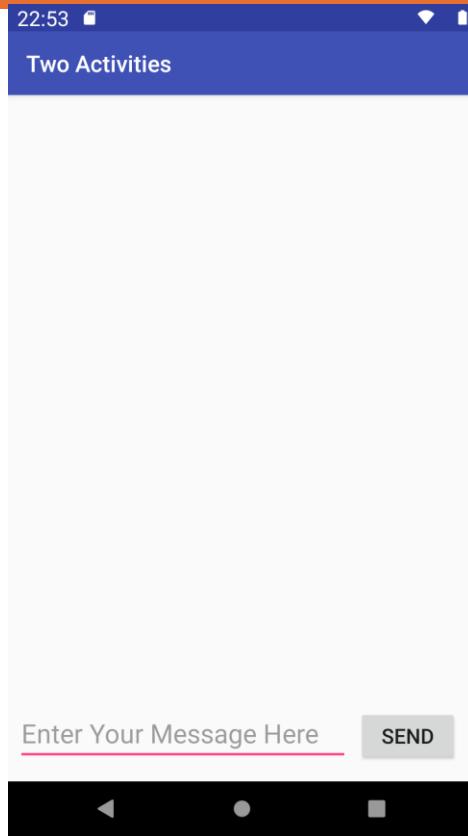
finish();
```

# Example Code

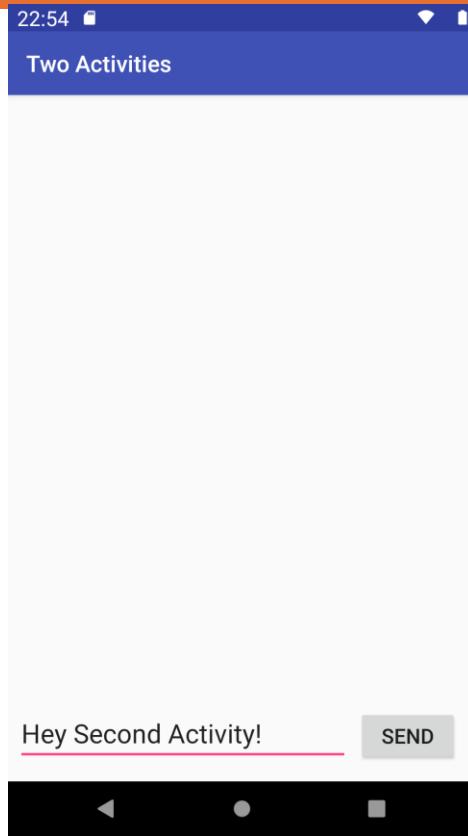
```
public void onActivityResult(int requestCode, int resultCode, Intent data)
{
    super.onActivityResult(requestCode, resultCode, data);

    if (requestCode == TEXT_REQUEST) {
        if (resultCode == RESULT_OK) {
            String reply = data.getStringExtra(SecondActivity.EXTRA_REPLY);
            // do something with the data
        }
    }
}
```

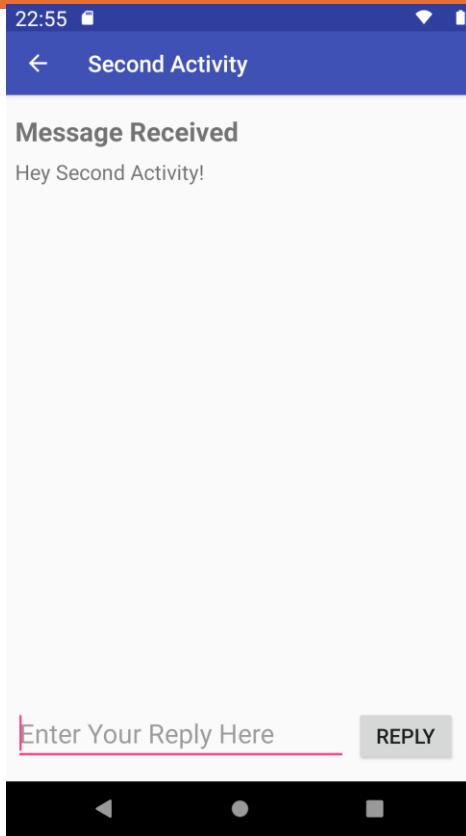
# Demo



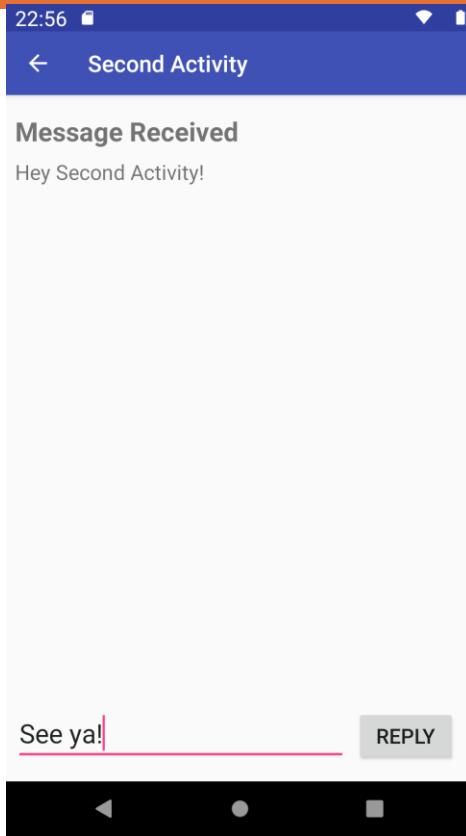
# Demo



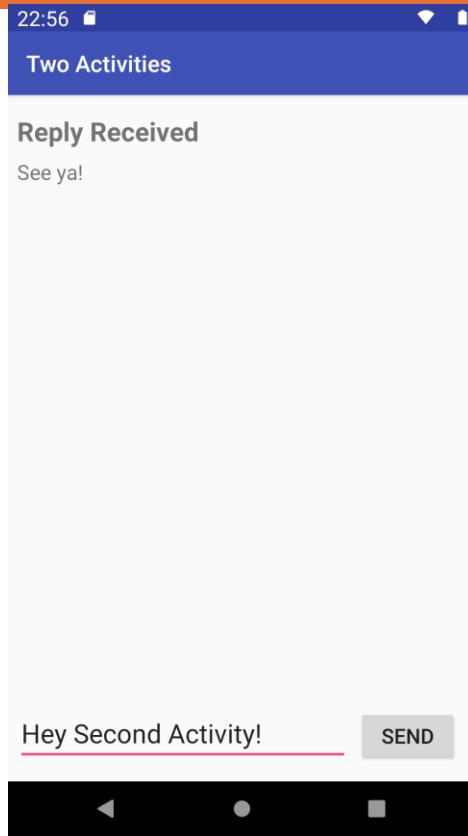
# Demo



# Demo



# Demo



# Navigation

# Activity Stack

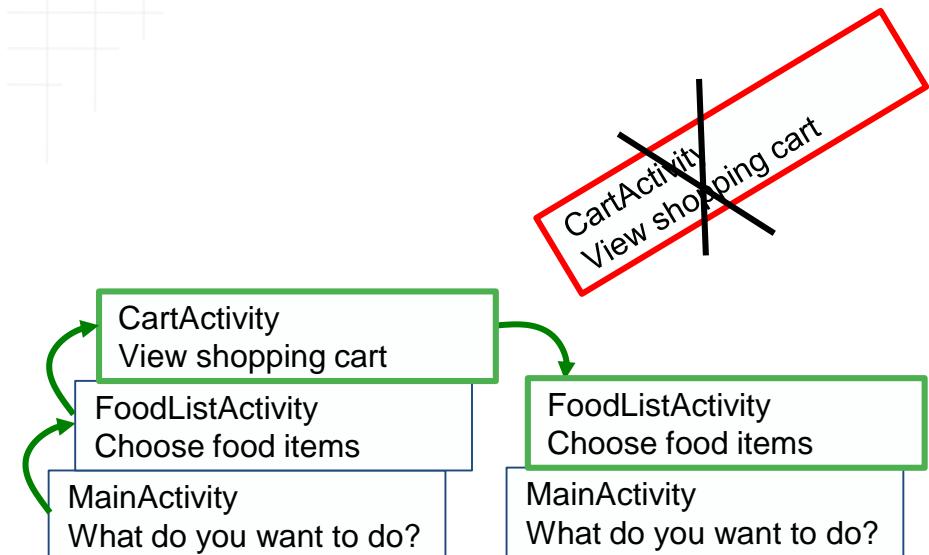
- When a new Activity is **started**, the previous Activity is **stopped** and pushed on the Activity back stack
- **Last In First Out** Stack - when the current Activity ends, or the user presses the Back button, it is popped from the stack and the previous Activity resumes

# Activity Stack



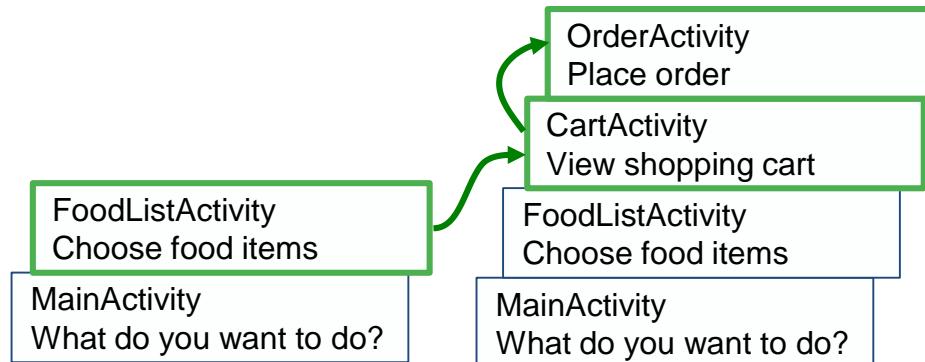
After viewing shopping cart, user decides to add more items, then places order.

# Activity Stack



After viewing shopping cart, user decides to add more items, then places order.

# Activity Stack



After viewing shopping cart, user decides to add more items, then places order.

# Activity Stack



After viewing shopping cart, user decides to add more items, then places order.

# Back Navigation



## Back navigation

- Provided by the device's Back button
- Controlled by the Android System's back stack

# Back Navigation

- Back Stack preserves history of recently viewed screens.
- Back Stack contains all the **Activity** objects that have been launched by the user in reverse order for the current task
- Each **task** has its own Back Stack
- Switching between tasks activates that task's Back Stack

# Handling Clicks

# One Way to Do It

- In XML use the android:onClick **attribute** in the XML layout:

```
<Button  
    android:id="@+id/button_main"  
    android:text="@string/button_main"  
    android:onClick="launchSecondActivity"  
/>
```

# One Way to Do It

- Then fill in the code **to handle** the click in **Java**

```
public void launchSecondActivity(View view) {  
  
    String message = mMessageEditText.getText().toString();  
  
    Intent intent = new Intent(this, SecondActivity.class);  
    intent.putExtra(EXTRA_MESSAGE, message);  
  
    startActivityForResult(intent, TEXT_REQUEST);  
}
```

# "HOLLYWOOD PRINCIPLE":



# DON'T CALL US, WE'LL CALL YOU

activity\_main.xml

```
<Button  
    android:id="@+id/button_main"  
    android:text="@string/button_main"  
    android:onClick="launchSecondActivity"  
  
/>
```

1<sup>st</sup> Activity

```
launchSecondActivity(View view)
```

```
{  
}
```

System calls  
**Callback** method!

Android  
System

# Mobile Software Development

**CMPU 3036 DT211C/3, DT282/3**

# Today's Lecture

- Manifest File
- Activity Lifecycle
  - **Callbacks**
  - Instance **State**
  - Saving and Restoring Activity State
- Quick **Demo**

# Manifest

SALOON, CABIN, AND STEERAGE ALIENS MUST BE COMPLETELY MANIFESTED.

LIST OR MANIFEST OF ALIEN PASSENGERS FOR THE UNITED

Required by the regulations of the Secretary of Commerce and Labor of the United States, under Act of Congress approved February 20, 1907, to be delivered

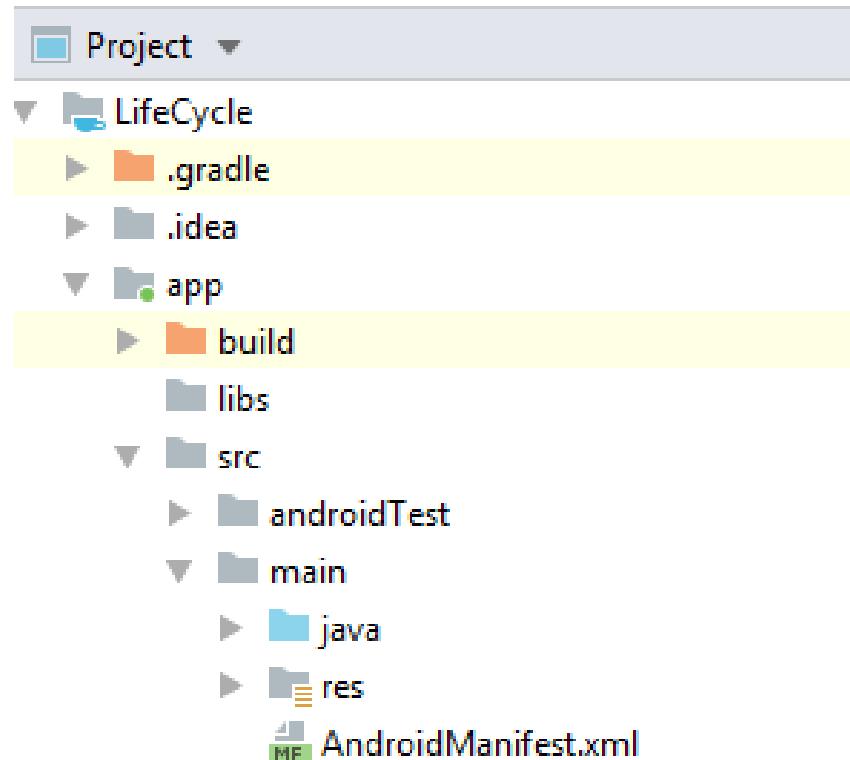
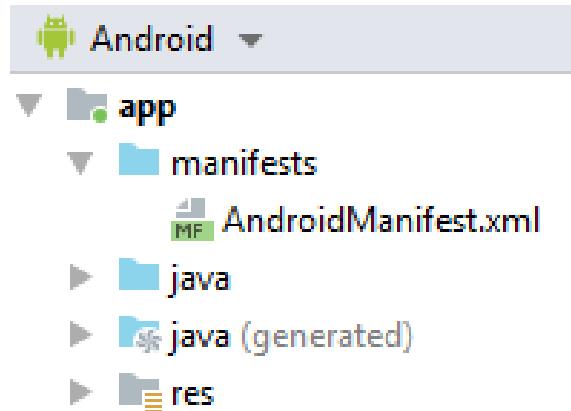
S. S. Priestland sailing from L'pool 9th November, 1910.

1	2	3	4	5	6	7	8	9	10	11	12
No. on List.	NAME IN FULL.	Age.	Sec.	Calling or Occupation.	Able to Read, Write.	Nationality. (Country of which citizen or subject.)	Race or People.	*Last Permanent Residence.	The name and complete address of nearest relative or friend in country whence alien came.	Final Destination. (Country where permanent residence.)	
	Family Name.	Given Name.	Yrs. Mth.					Country.	City or Town.	State.	City or Town.
163 ✓	Witkunec	John	19	Lab. (laborer)	-	Austrian	Polish	mother Mrs. Agnes R. Ellsworth			
164 ✓	Pusk	Polyneby	16	an L. P. (laborer)	Russian	Russian	Polish	John W. Johnson	Hornell		
165 ✓	Rudinski	Jurgis	23	Lab. (laborer)	Polish	Polish	Polish	NOV 17 1922	VERMONT	11/23/1923	
166 ✓	Makowski	Frank	22	Lab. (laborer)	Polish	Polish	Polish	W. I. T. Newell	Gilberton		
167 ✓	Leukiewicz	Era	16	Lab. (laborer)	Polish	Polish	Polish	Teller in Newell	Newell	Cyrus Farren	
168 ✓	Koplawski	Matus	18	Lab. (laborer)	Polish	Polish	Polish	Samuel Goldfarb	Grandville		
169 ✓	Klausner	Mindel	14	Lab. (laborer)	Polish	Polish	Polish	Eliezer Goldfarb	Blatow		
170 ✓	Meische	114	1 m	Lab. (laborer)	Polish	Polish	Polish	David Goldfarb	Cleveland	Ohio Cleveland	
171 ✓		70473	MAY 1	10 1 d. Child	Polish	Polish	Polish	de			
172 ✓		15051		1.8 m - FEB 20 1931	Polish	Polish	Polish	de			
173 ✓		8682	1/23/31	Rueke	1 1/2 d. Inf.	Polish	Polish	de			

# Intro

- Every app project **must** have an `AndroidManifest.xml` file.
- It lives at the **root** of the project source set.
- Describes **essential** information about your app to:
  1. Android Build Tools
  2. Android Operating System
  3. Google Play

# Location



# Rules

1. Only <manifest> and <application> **elements** are required.
  - Must occur only **once**.
2. Most other elements can occur 0 or more times.
3. All values are set through **attributes**.
4. Elements at same level are generally **not** ordered.

# Example

```
<?xml version="1.0" encoding="utf-8"?>
<manifest >
    <application
        android:theme = "@style/AppTheme">
        <activity android:name = ".ActivityA">
            <intent-filter>
                </intent-filter>
        </activity>
        <activity />
        <activity />
    </application>
</manifest>
```

# Contents

Things found inside manifest file include:

1. Package name
  - Used to find pieces of code when **building** your project.
2. Components e.g. Activities
  - Each component must define basic properties e.g. name of **Java** class.

# Contents

## 3. Capabilities

- Device configurations it can handle and Intent filters (start-up).

## 4. Permissions

- For App to access **protected** parts of the system or other apps.

## 5. Features required

- Affects which devices can **install** the app from Google Play

# Package Name

- The manifest file's **root** element requires an **attribute** for your app's package name:

```
<manifest  
    xmlns:android = "http://schemas.android.com/apk/res/android"  
    package = "com.example.lifecycle">
```

# Package Name

- When building the APK, the package attribute is important e.g. as **namespace** for generated R class com.example.lifecycle.R
- It also resolves **relative** class names declared in the manifest file e.g. com.example.lifecycle.ActivityA

```
<activity android:name=".ActivityA">
```

# Permissions

- Android apps must **request** permission to access:
  - **Sensitive** user data (contacts and SMS)
  - Certain system features (**camera** and network connection)
- Permissions are requested with a `<uses-permission>` **element** and identified by a unique label e.g. to send SMS messages

```
<uses-permission android:name = "android.permission.SEND_SMS"/>
```

# Permissions

- Beginning with API level 23, the **user** can approve or reject some app permissions at **runtime**.
- If the permission is **granted**, the app is able to use the protected features. If not, its attempts to access those features **fail**.

# App Components

- For each app component in your app, declare an XML **element**:

Activity	<activity>
Service	<service>
BroadcastReceiver	<receiver>
ContentProvider	<provider>
- If you **subclass** these components without declaring it in the manifest file, the system cannot start it.

# App Components

- When an app issues an **Intent** to the system, it locates an app component that can handle it based on `<intent-filter>` declarations in each app's manifest file.

# Device Compatibility

- Play Store does **not** allow app be installed on devices that don't provide required features.
- Hardware or software features your app requires are declared with **<uses-feature> element**.

```
<uses-feature android:name = "android.hardware.sensor.compass"  
            android:required = "true" />
```

# Activity Lifecycle

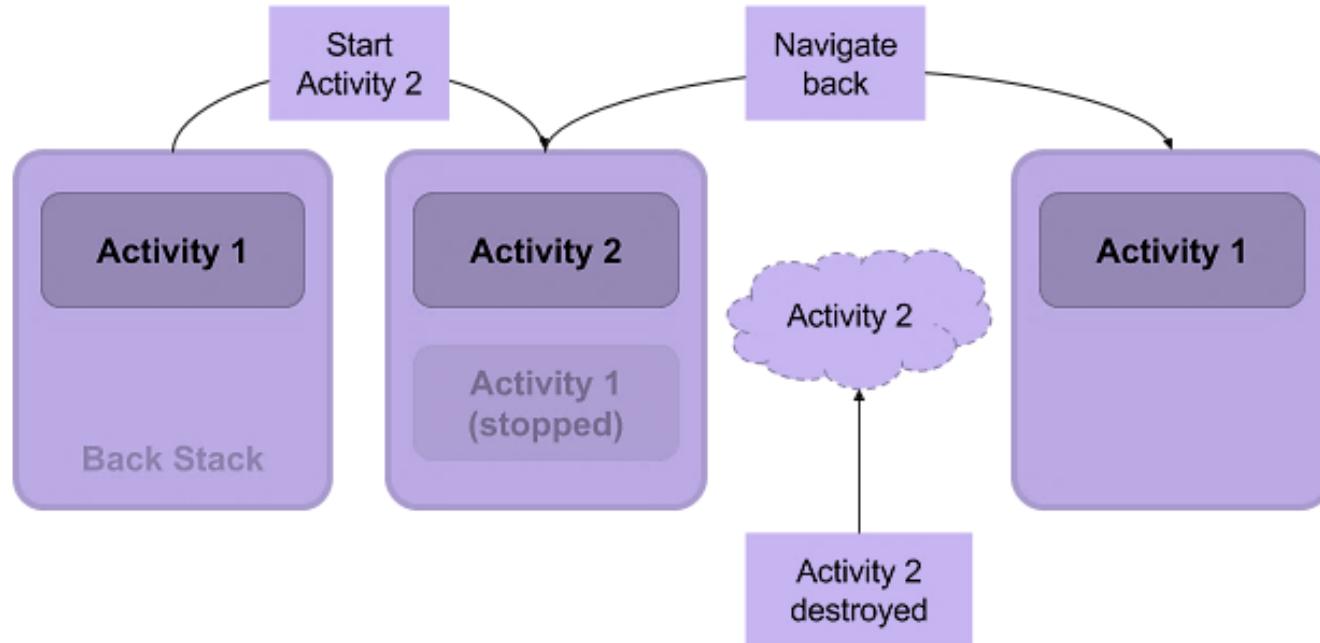
# What is the Activity Lifecycle?

- The set of **states** an Activity can be in during its lifetime, from when it is created **until** it is destroyed

In Google Speak:

- A directed graph of all the states an Activity can be in, and the **callbacks** associated with transitioning from each state to the next one

# What is the Activity Lifecycle?



# States

Lifecycle state changes are **triggered** by user action, configuration changes such as device rotation, or system action.

# Activity Lifecycle Callbacks

# Implementing Callbacks

- Only `onCreate()` is **required**.
- Override the other callbacks to change **default** behaviour.

# On Create

```
@Override  
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
  
    // Activity is being created.  
}
```

# On Create

- Called when Activity is first created e.g. user hits **launcher** icon.
- Does all **static** setup: create views, bind data to lists...
- Only called **once** during an activity's lifetime.
- Takes a **bundle** with Activity's previously frozen state, if exists.
- Created state is always followed by onStart()

# On Start

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

# On Start

- Called when the Activity is becoming **visible** to user.
- Can be called **more than once** during lifecycle.
- Followed by onResume() if the activity comes to the foreground, or onStop() if it becomes hidden.

# On Restart

```
@Override  
protected void onRestart() {  
    super.onRestart();  
  
    // Activity is between stopped and started.  
}
```

# On Restart

- Called after Activity has been **stopped**, immediately before it is started again.
- Transient state.
- Always followed by onStart().

# On Resume

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

# On Resume

- Called when Activity will start **interacting** with user.
- Activity has moved to top of the **Activity Stack**.
- Starts accepting **user** input.
- Running state.
- Always followed by onPause().

# On Pause

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

# On Pause

- Called when system is about to resume a previous Activity.
- Activity is partly visible but user is leaving the Activity.
- Typically used to commit unsaved changes to persistent data, stop animations and anything that **consumes resources**.

# On Pause

- Implementations must be fast because **next** Activity is **not** resumed **until** this method **returns**.
- Followed by either onResume() if the Activity returns back to the front or onStop() if it becomes invisible to the user.

# On Stop

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

# On Stop

- Called when the Activity is no longer **visible** to the user.
- A **new** Activity is being started, an existing one is brought in front of this one, or this one is being destroyed.
- Operations that were too heavy-weight for onPause()
- Followed by either onRestart() if Activity is **coming back** to interact with user, or onDestroy() if Activity is **going away**.

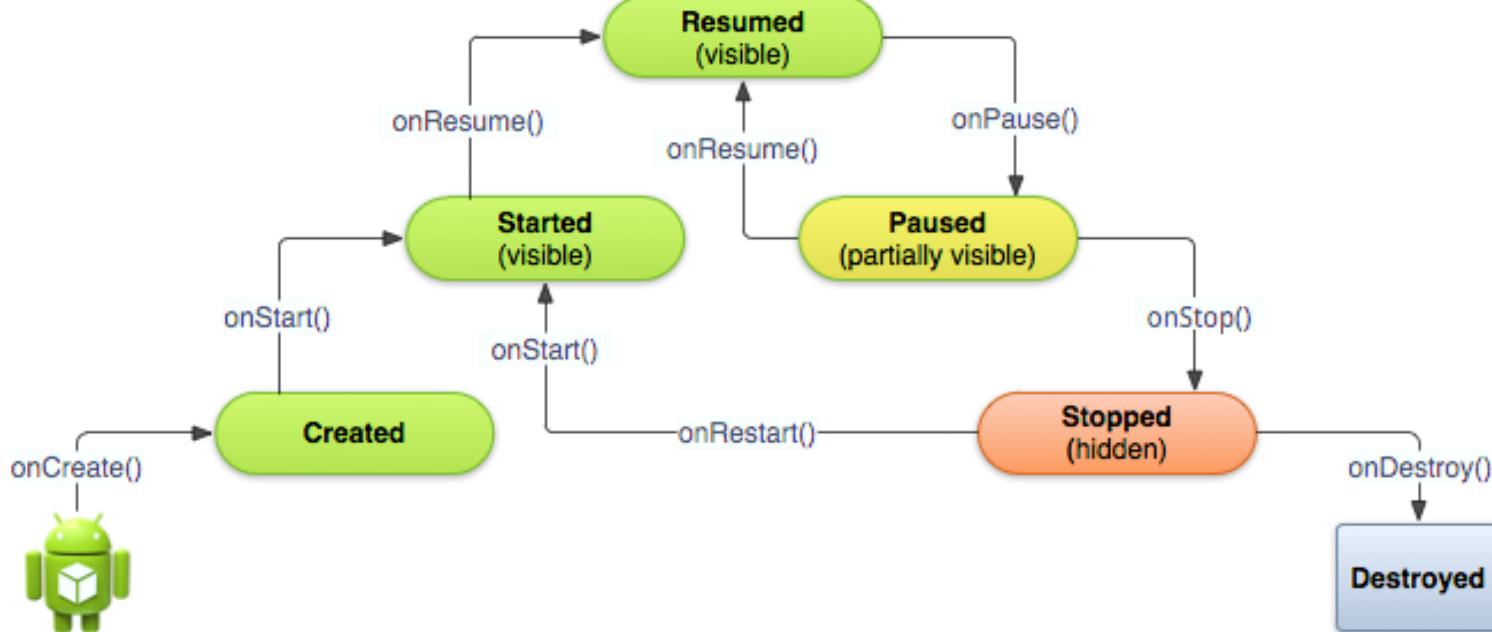
# On Destroy

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

# On Destroy

- **Final** call before Activity is destroyed.
- User navigates **back** to previous Activity, or configuration changes.
- Activity is **finishing** or system is destroying it to save space.
- Call `isFinishing()` method to check.
- The **system** may destroy Activity without calling this, so use `onPause()` or `onStop()` to save data or state.

# States and Callbacks Graph



# App Visibility

- Created (not visible yet)
- Started (visible)
- Resume (visible)
- Paused(partially invisible)
- Stopped (hidden)
- Destroyed (gone from **memory**)

# Summary

**onCreate()** - Static initialisation

**onStart()** - Activity is becoming visible

**onRestart()** - If Activity was stopped (calls onStart())

**onResume()** - Start to interact with user

**onPause()** - About to resume previous Activity

**onStop()** - No longer visible, but still exists and **all state** info preserved

**onDestroy()** - Final call before System destroys Activity

# Example User Behaviour



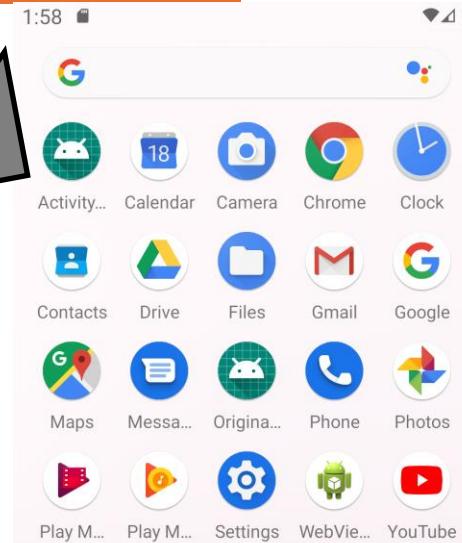
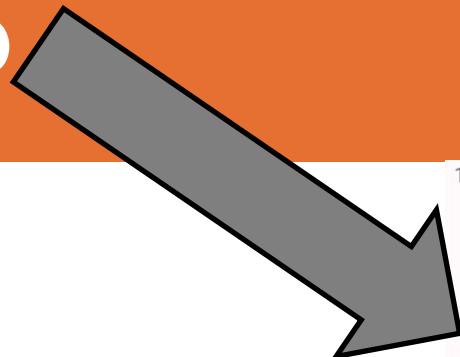
# User Launches App

Activity A Launched

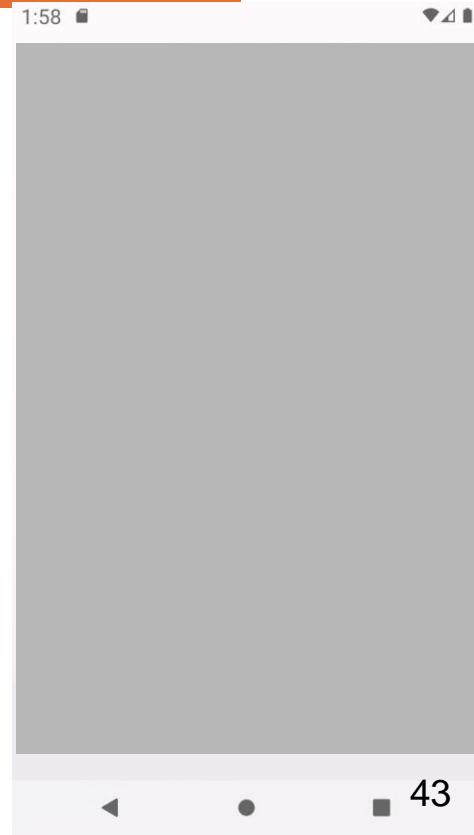
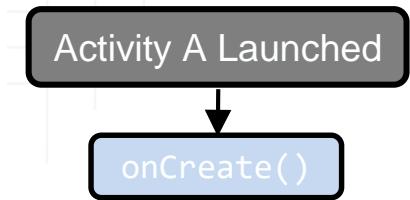
```
<activity android:name=".ActivityA">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

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

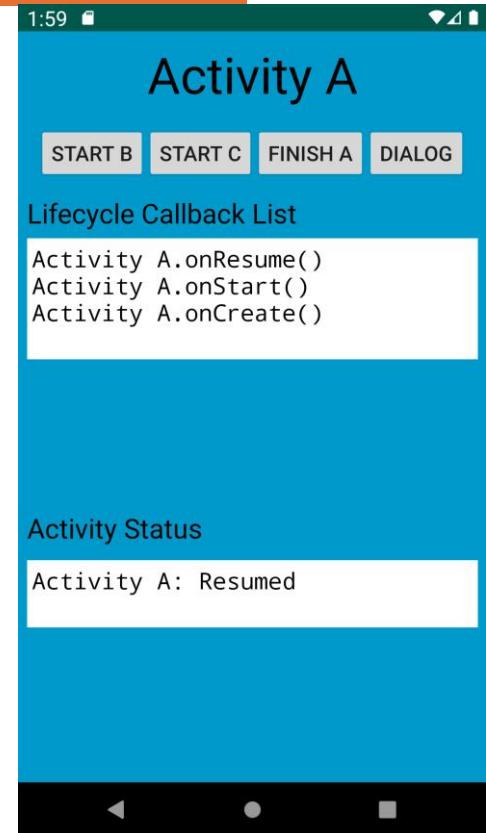
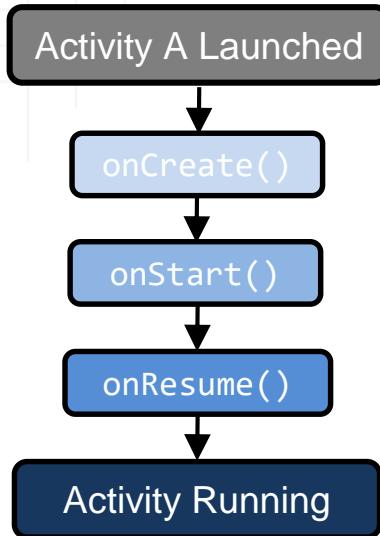
AndroidManifest.xml



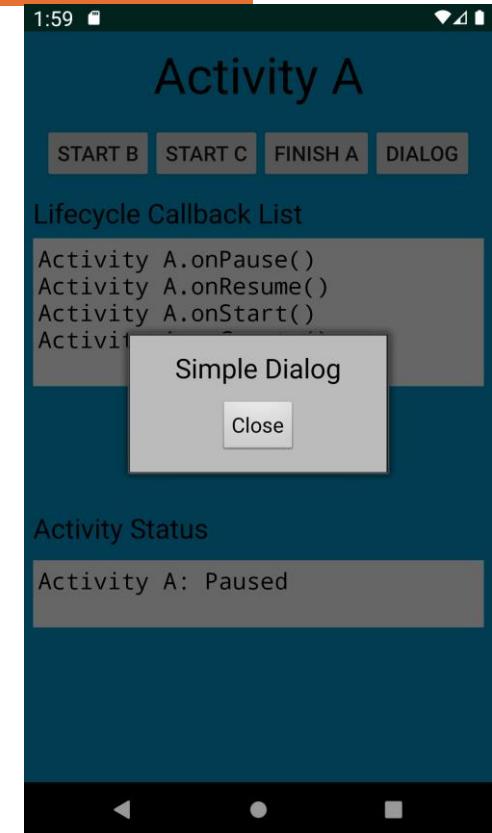
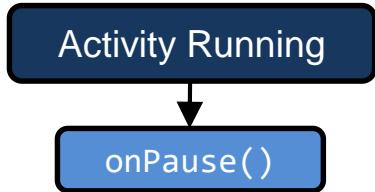
# User Launches App



# User Launches App



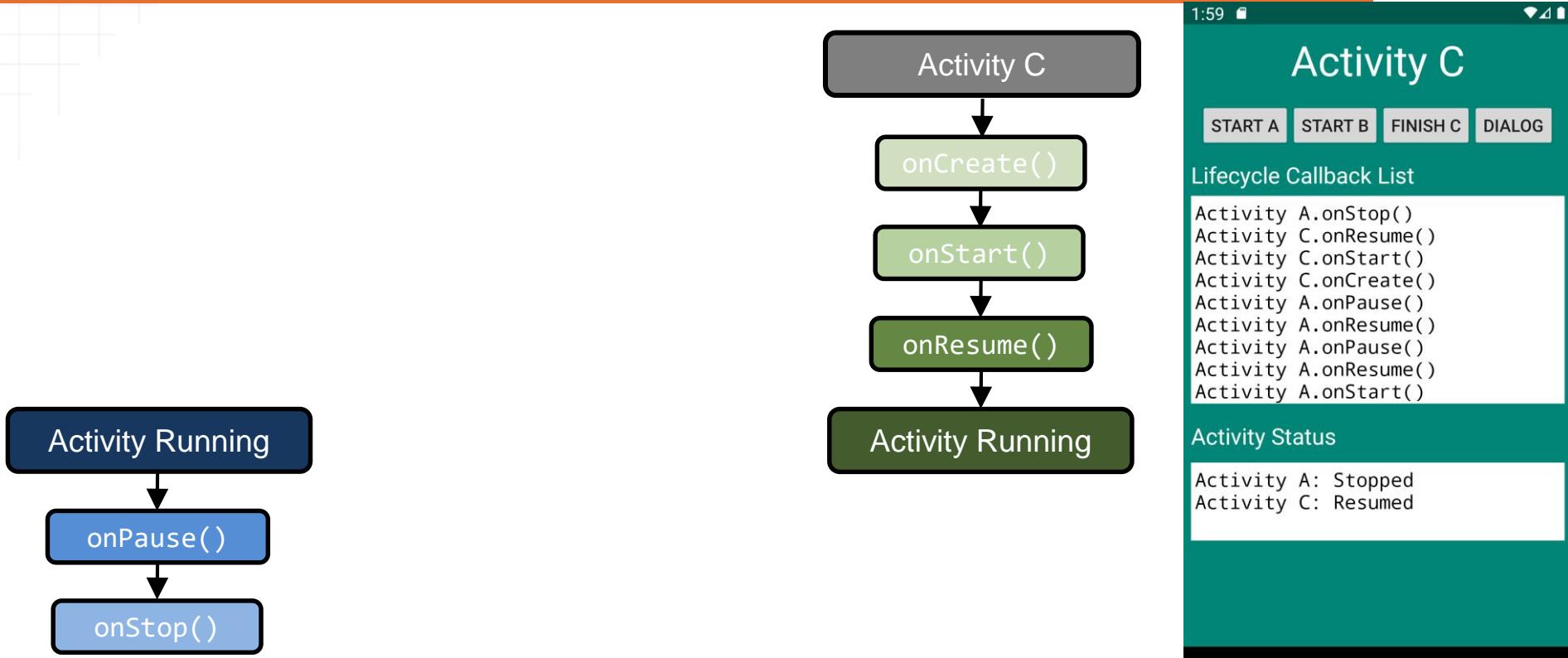
# User Clicks Dialog Button



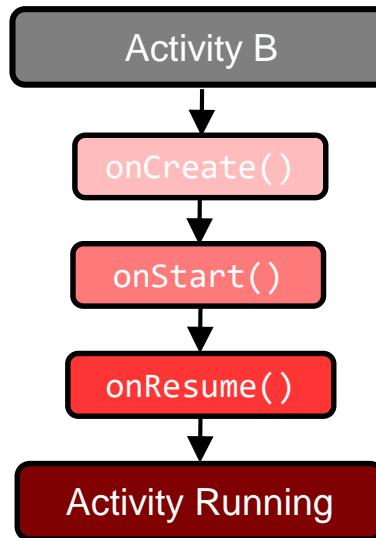
# User Closes Dialog



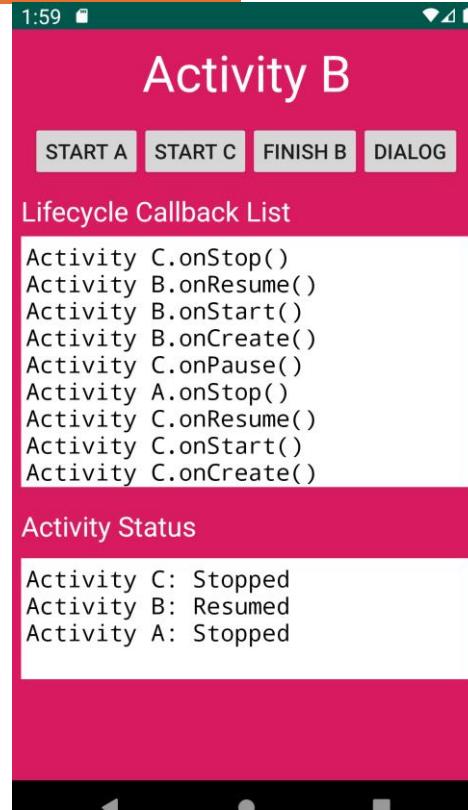
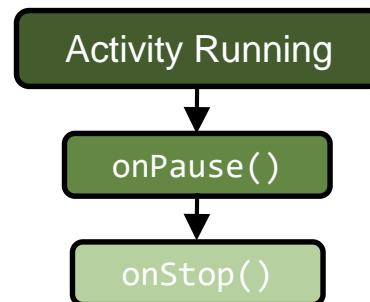
# User Starts Activity C



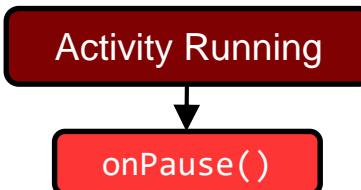
# User Starts Activity B



onStop()

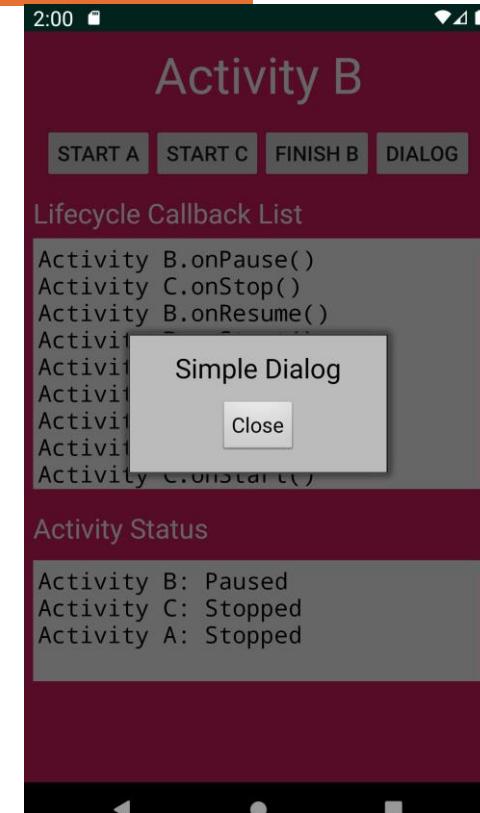


# User Clicks Dialog Button

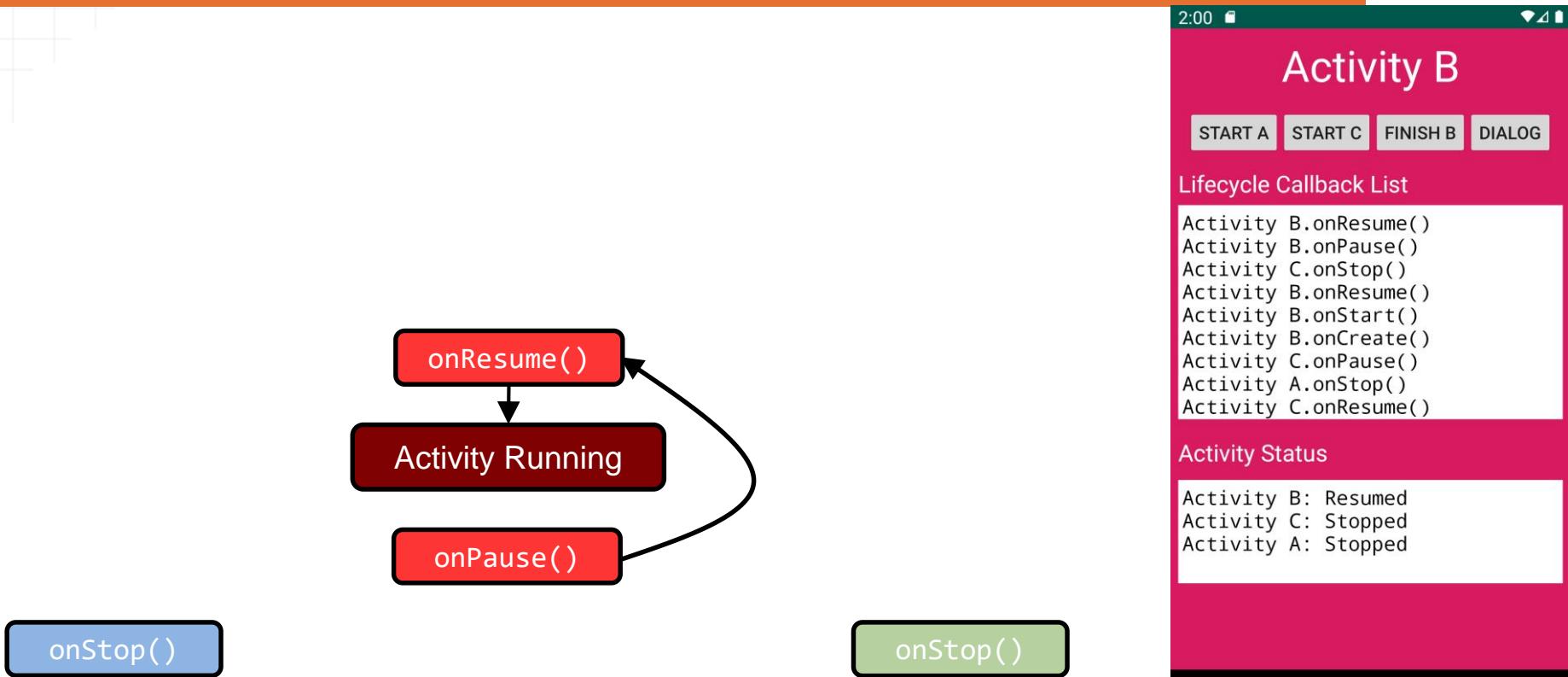


`onStop()`

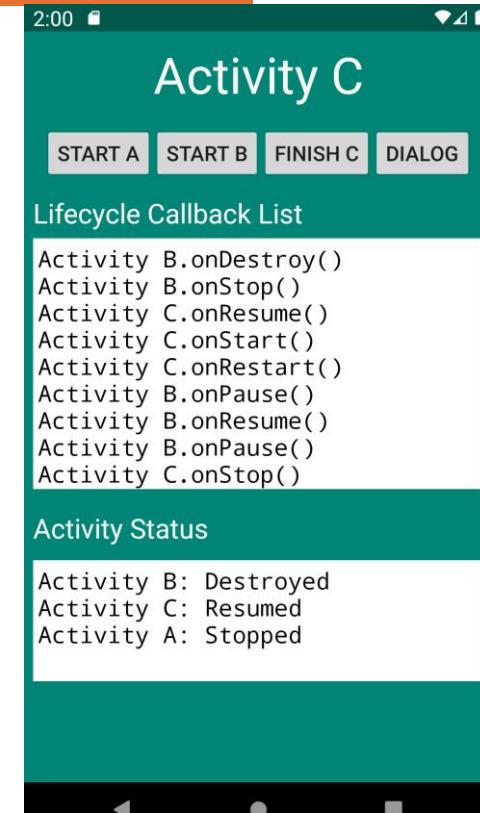
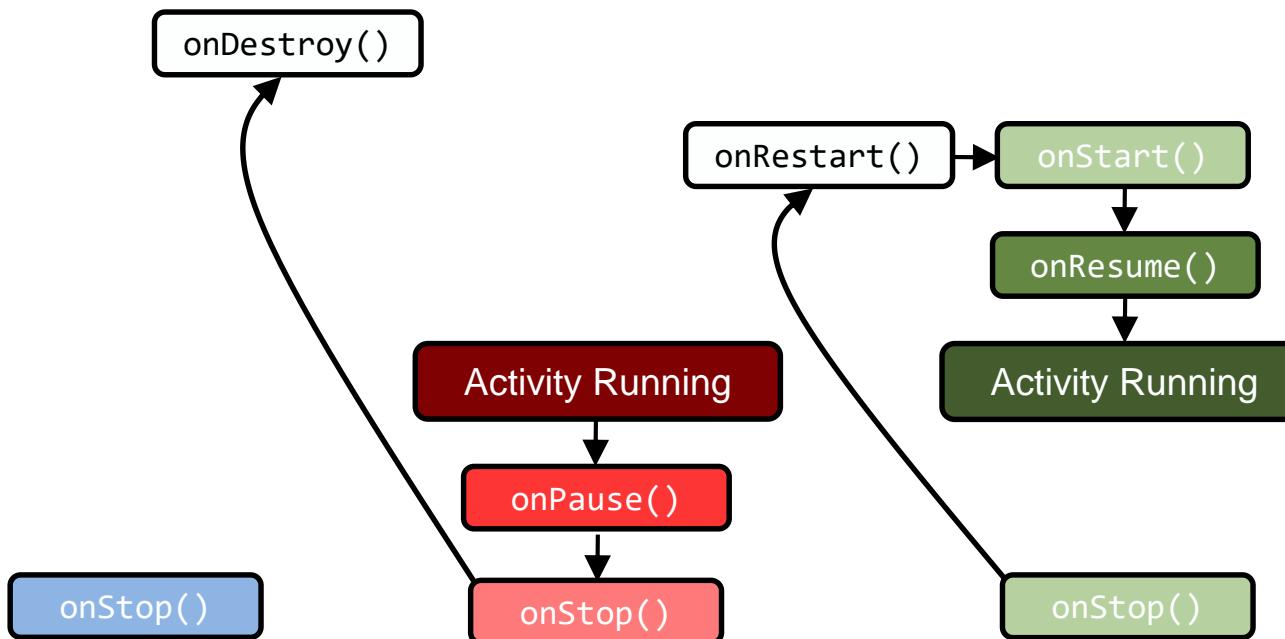
`onStop()`



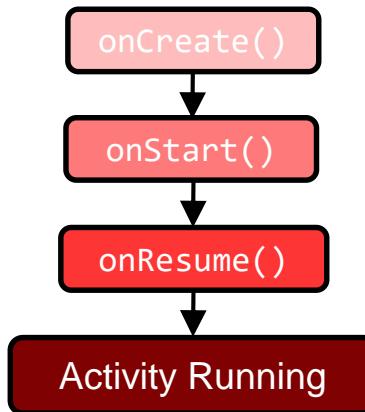
# User Closes Dialog



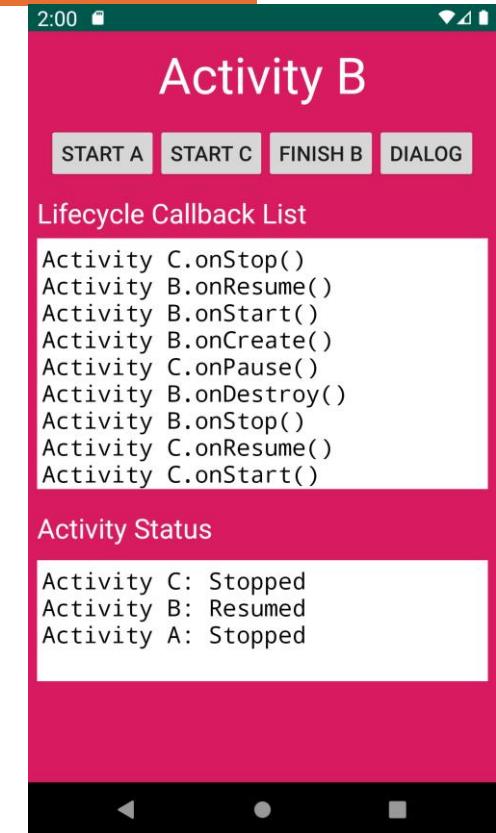
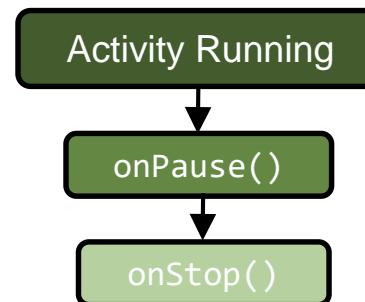
# User Hits Back Button



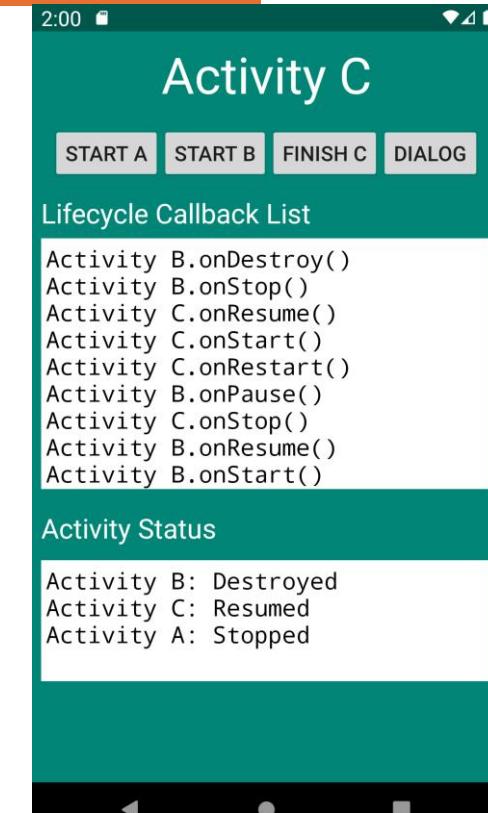
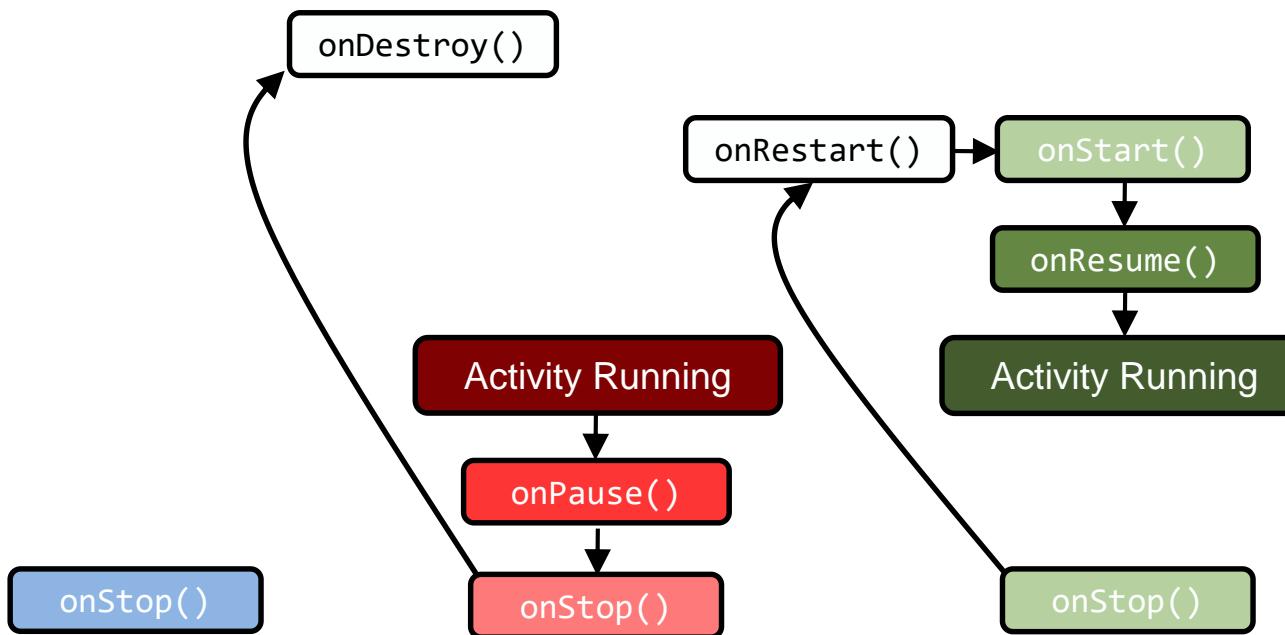
# User Starts Activity B



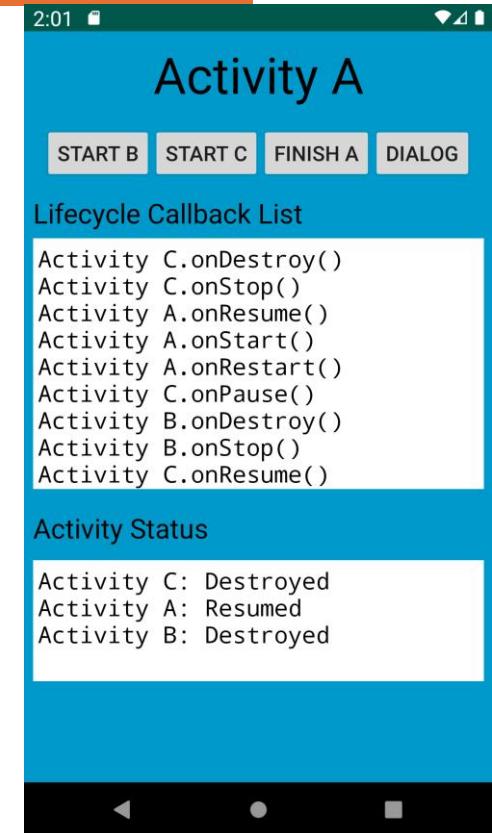
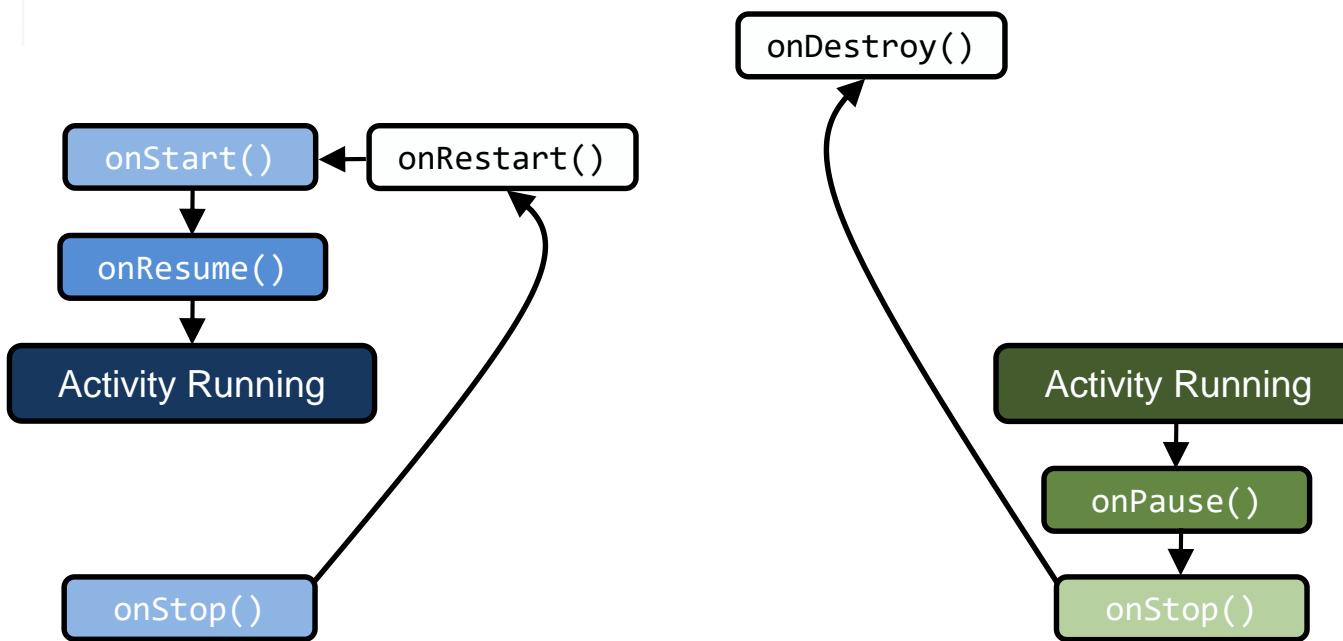
onStop()



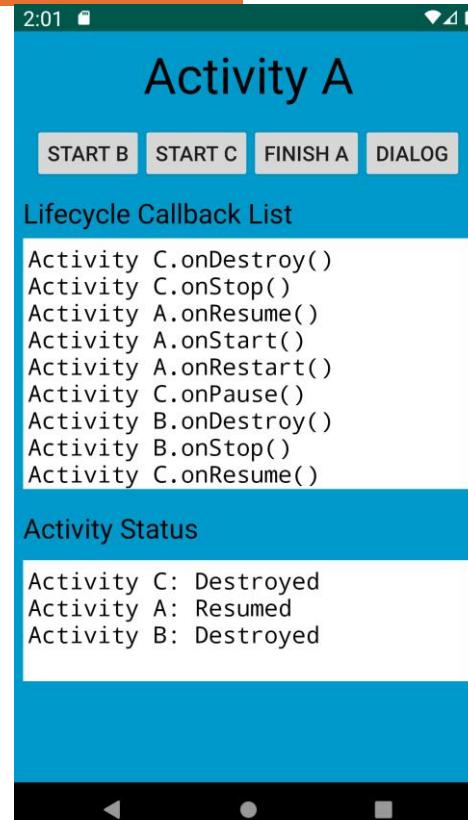
# User Hits Back Button



# User Hits Back Button



# Example Ends



Activity Running

# Activity Instance State

# When does Config Change?

Configuration changes **invalidate** the current layout or other resources in your activity when the user:

- **Rotates** the device
- Chooses different system language, so **locale** changes
- Enters **multi-window** mode (from Android 7)

# On Config Change...

On configuration change, Android:

1. Shuts down Activity by calling:
  - `onPause()`, `onStop()` and `onDestroy()`
2. Starts Activity over again by calling:
  - `onCreate()`, `onStart()` and `onResume()`

# Activity Instance State

- State information is created while the Activity is running e.g. a counter, user text, animation progression.
- State is **lost** when
  1. Device is **rotated**
  2. Language **changes**
  3. Back Button is pressed
  4. System **clears** memory

# Saving and Restoring Activity State

# What the System Saves

- System saves only:
  - **State** of views with unique ID (android:id) such as text entered into EditText
  - **Intent** that started activity and data in its extras
- **You** are responsible for saving other activity and user progress data

# Saving Instance State

Implement `onSaveInstanceState()` in your Activity

- Called by Android System when there is a possibility the Activity may be **destroyed**.
- Saves data only for this instance of the Activity during current session.

# Saving Instance State

```
@Override  
public void onSaveInstanceState(Bundle outState) {  
    super.onSaveInstanceState(outState);  
  
    // Add information for saving a counter to the outState bundle  
    outState.putString("count", String.valueOf(mShowCount.getText()));  
}
```

# Restoring Instance State

2 ways to retrieve the saved Bundle

- In `onCreate(Bundle mySavedState)`
  - Preferred, to ensure that your user interface, including any saved state, is back up and running as quickly as possible
- Implement callback (called after `onStart()`)  
`onRestoreInstanceState(Bundle mySavedState)`

# Restoring on Creation

```
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
    mShowCount = findViewById(R.id.show_count);
```

```
if (savedInstanceState != null) {  
    String count = savedInstanceState.getString("count");  
    if (mShowCount != null)  
        mShowCount.setText(count);  
}
```

# Restoring Instance State

```
public void onRestoreInstanceState (Bundle mySavedState) {  
    super.onRestoreInstanceState(mySavedState);
```

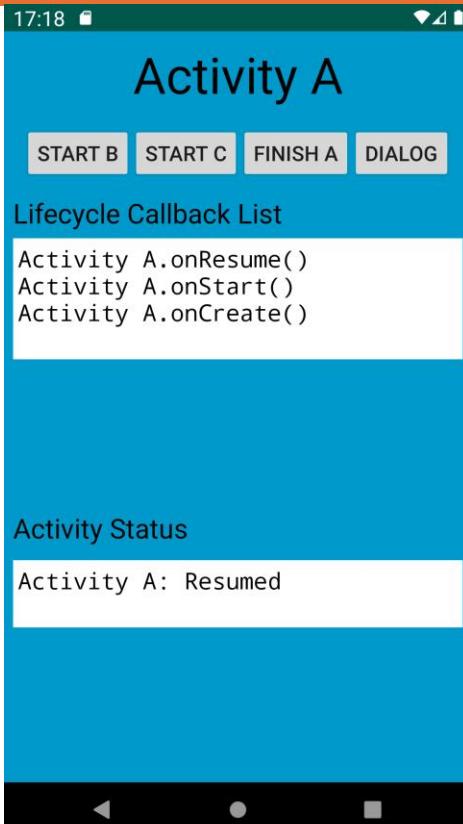
```
    if (mySavedState != null) {  
        String count = mySavedState.getString("count");  
        if (count != null)  
            mShowCount.setText(count);  
    }}  
}
```

# InstanceState and App Restart

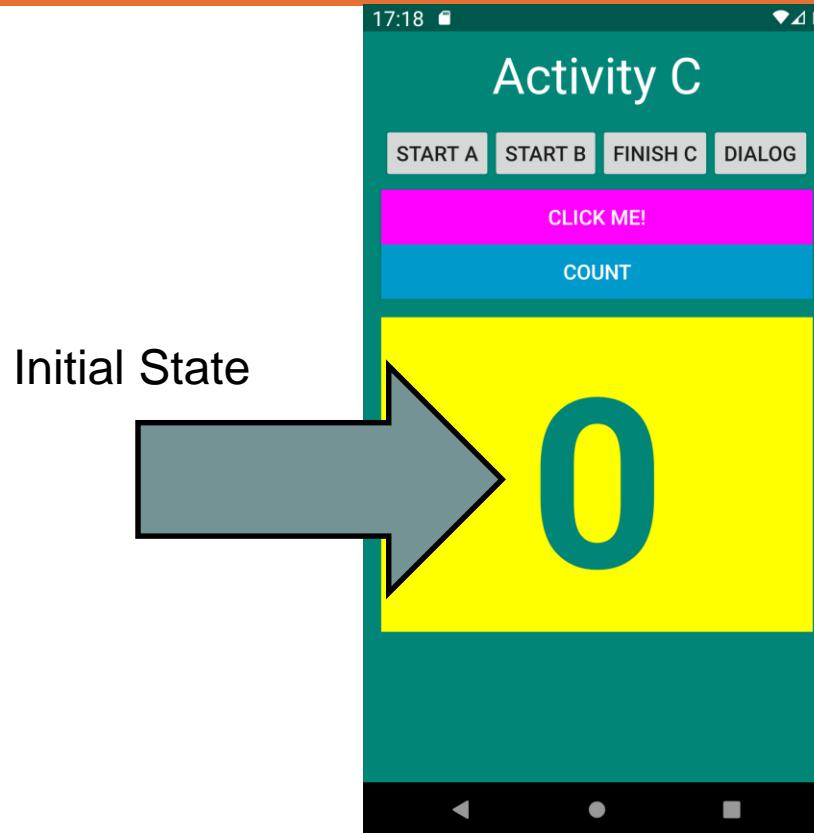
When you stop and restart a new app session, the Activity instance states are lost and your activities will revert to their default appearance

If you need to save (persist) user data between app sessions, use shared preferences or a **database**.

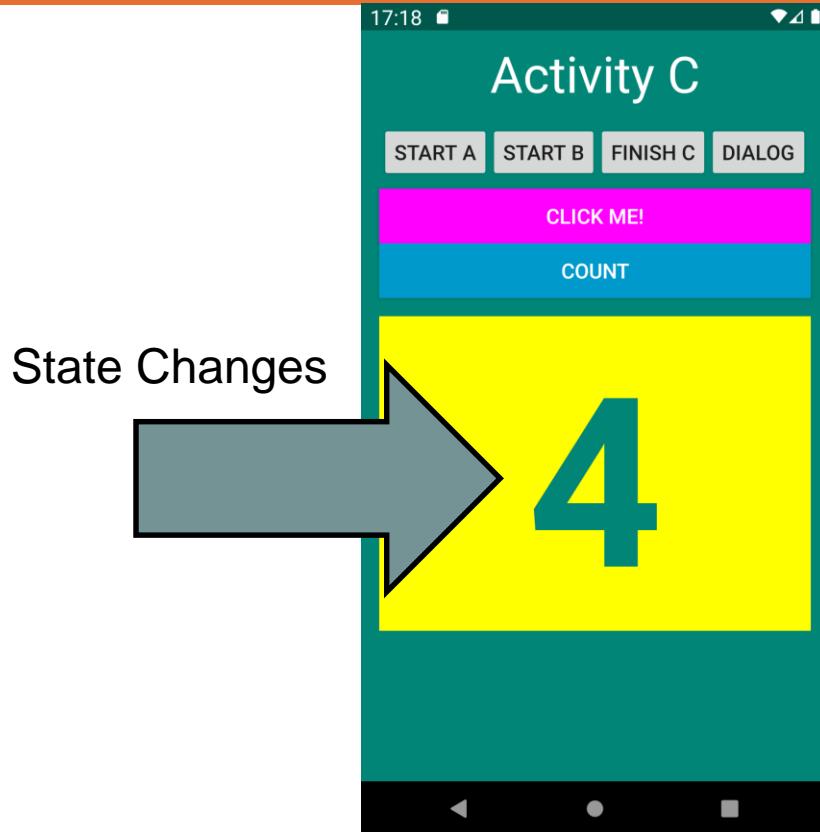
# Launch Activity



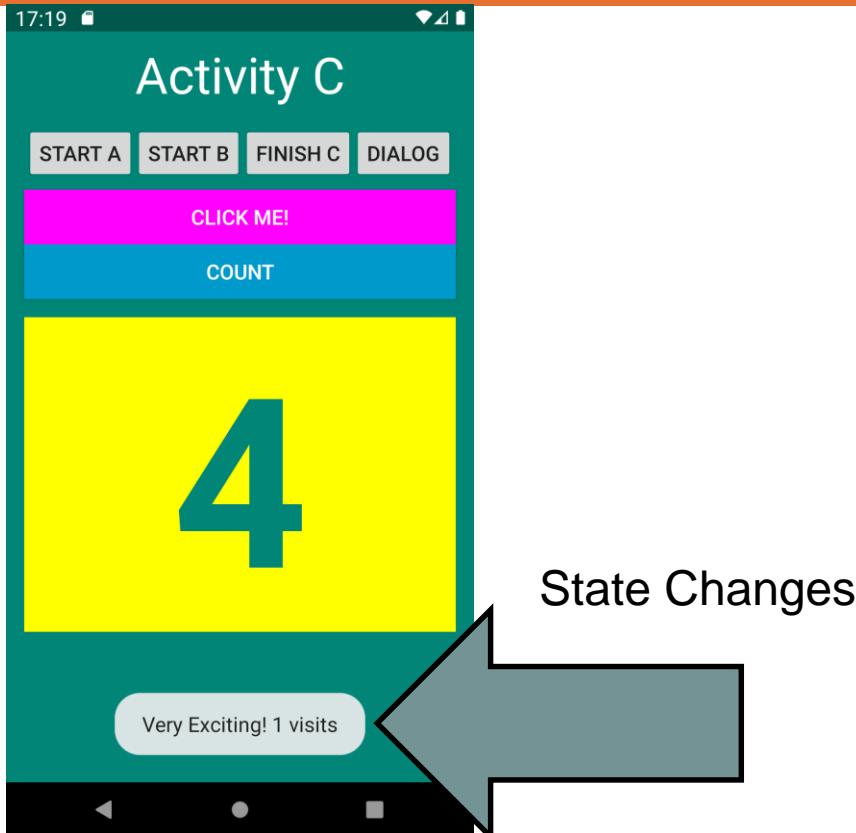
# User Launches Activity C



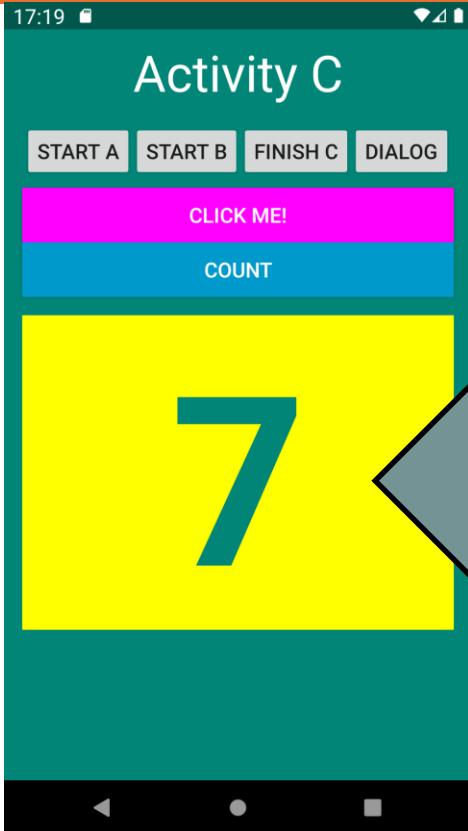
# Button Click Counts



# Screen Visits

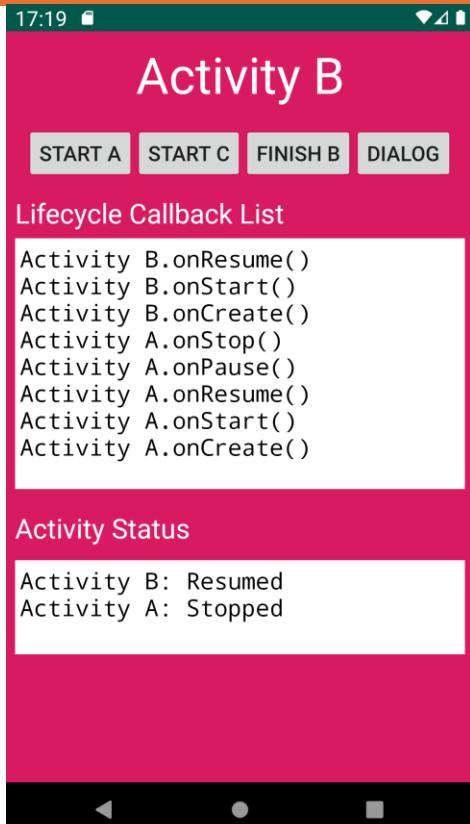


# Button Click Counts

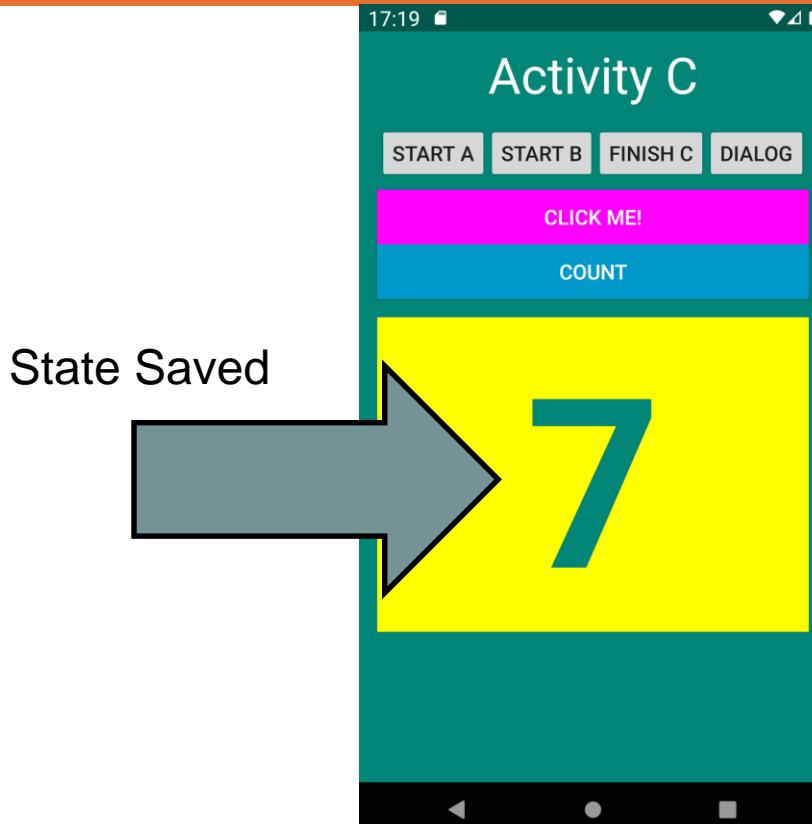


State Changes

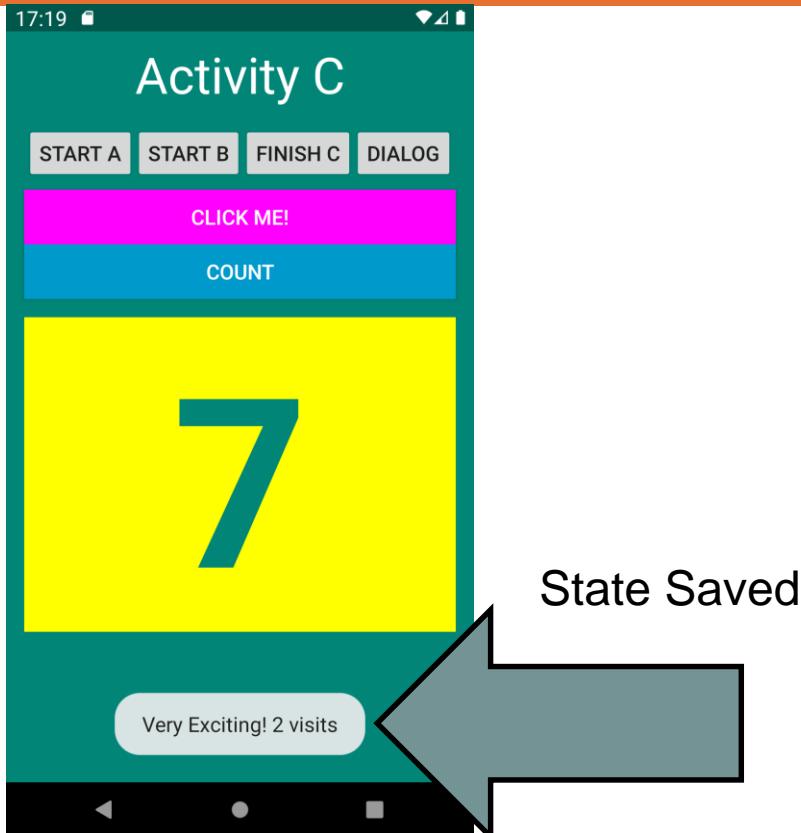
# Launch Activity B



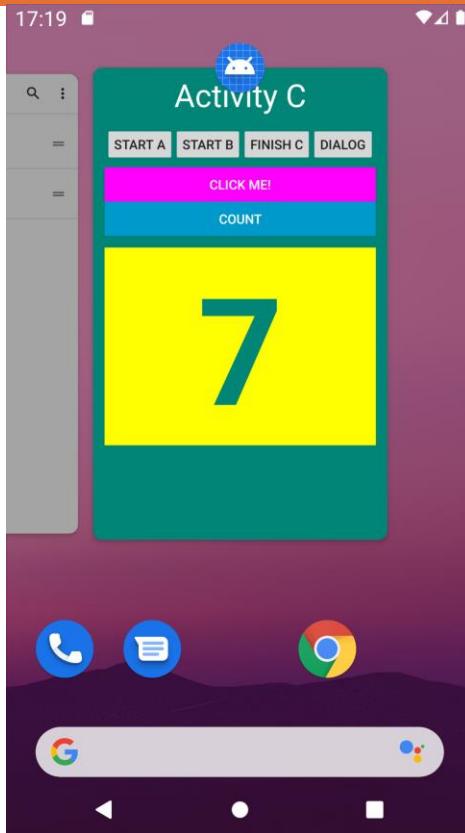
# Click Back Button



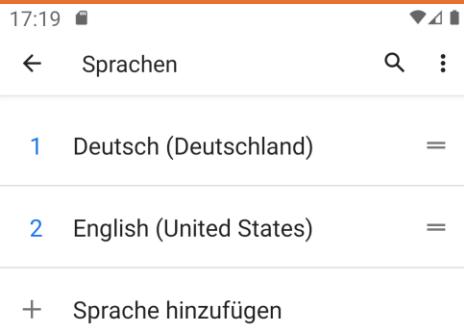
# Screen Visits



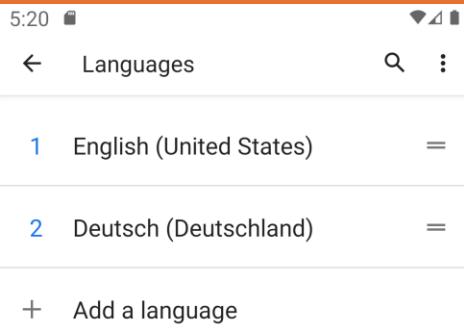
# User Leaves App



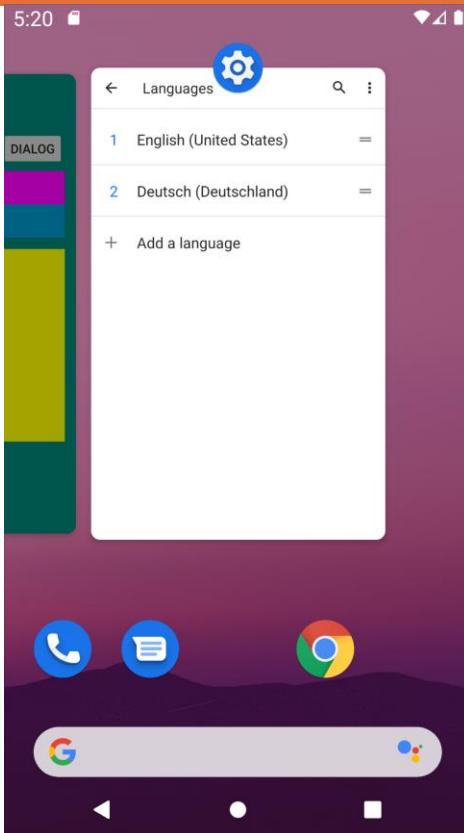
# Configuration Change



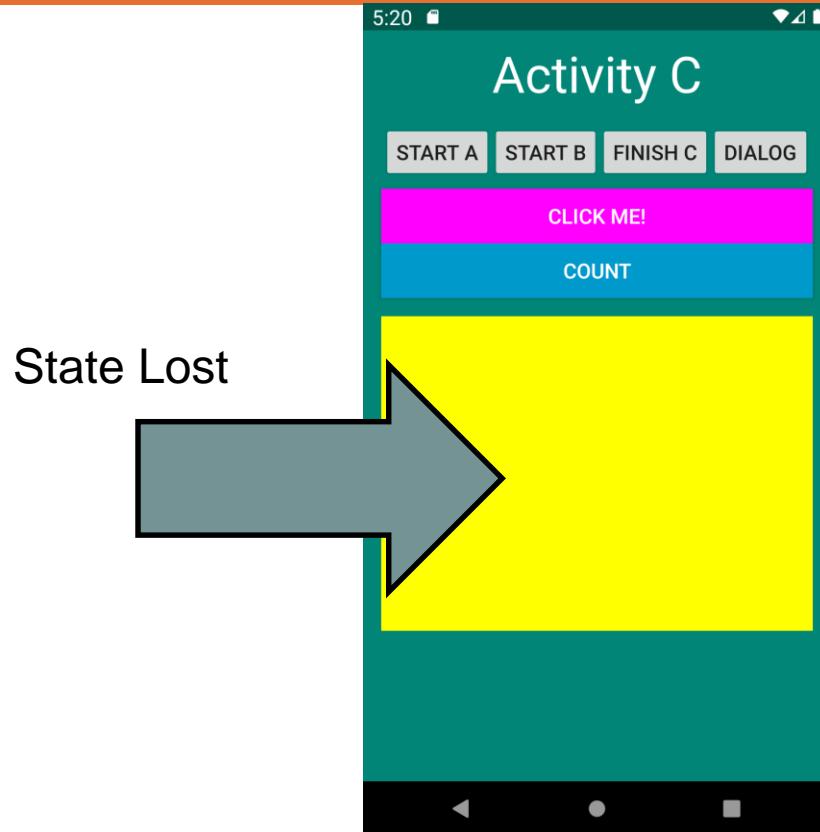
# New Language



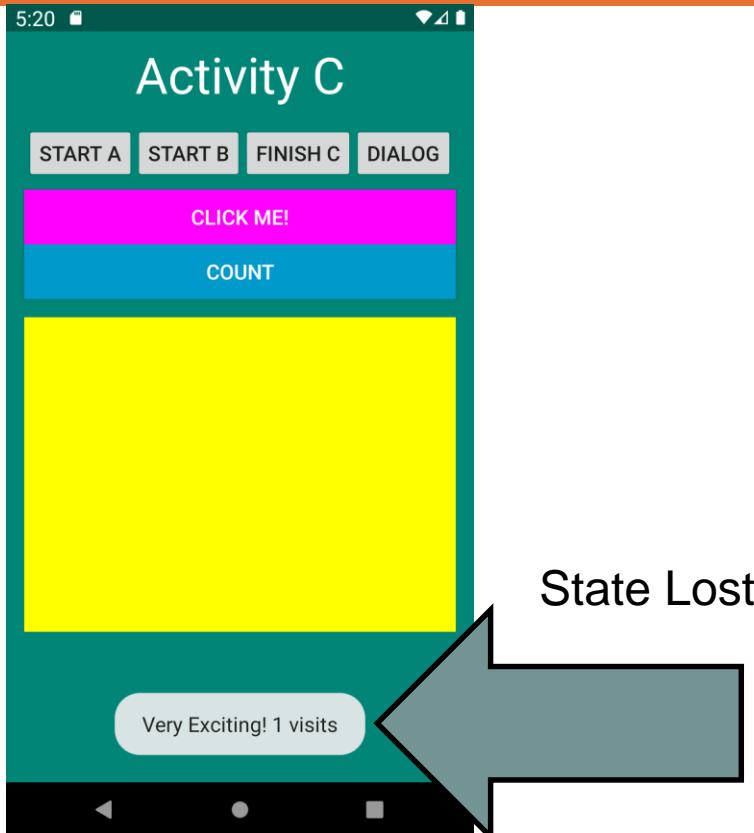
# User Returns to App



# User Returns to App



# User Returns to App

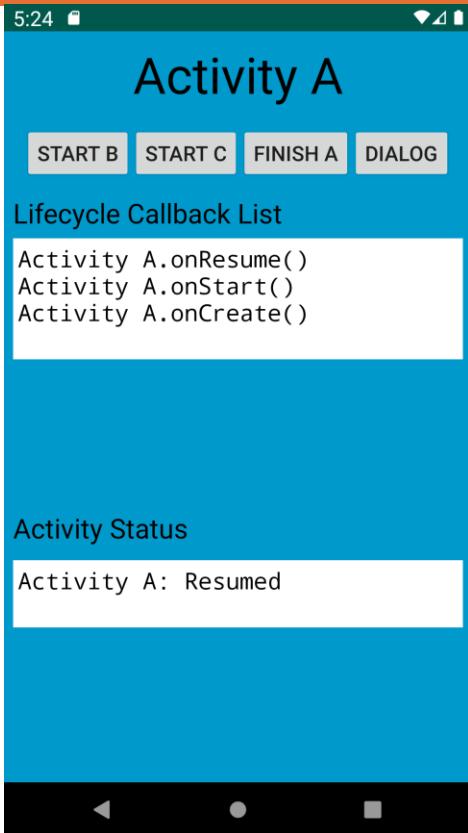


# Implement Save State

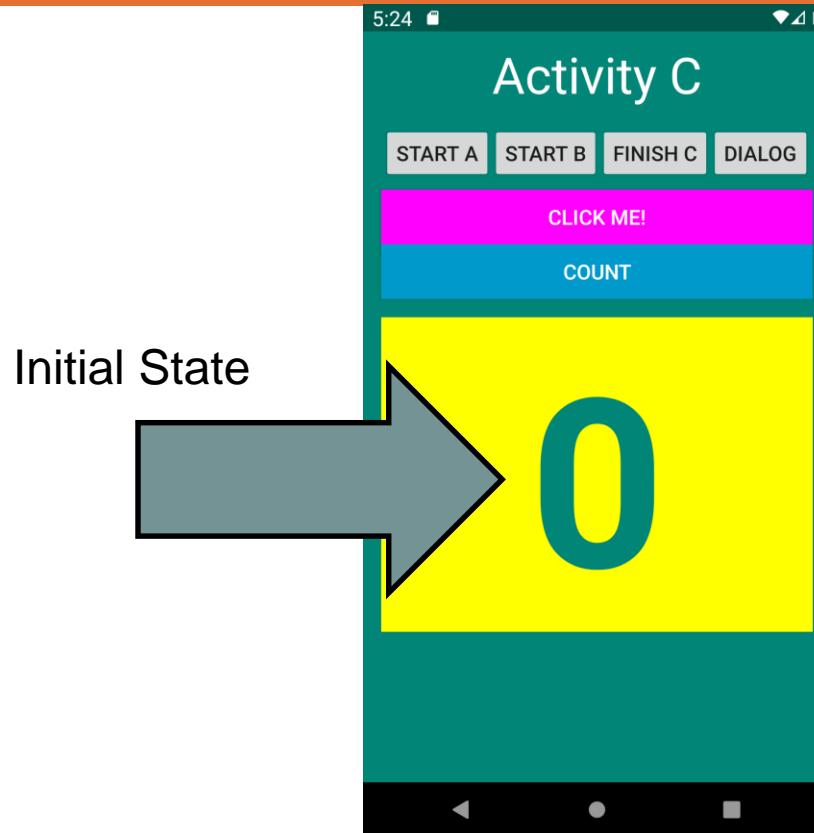
```
@Override
    public void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);

        outState.putString("count", String.valueOf(mShowCount.getText()));
        outState.putInt("visits", mVisits);
        super.onSaveInstanceState(outState);
    }
}
```

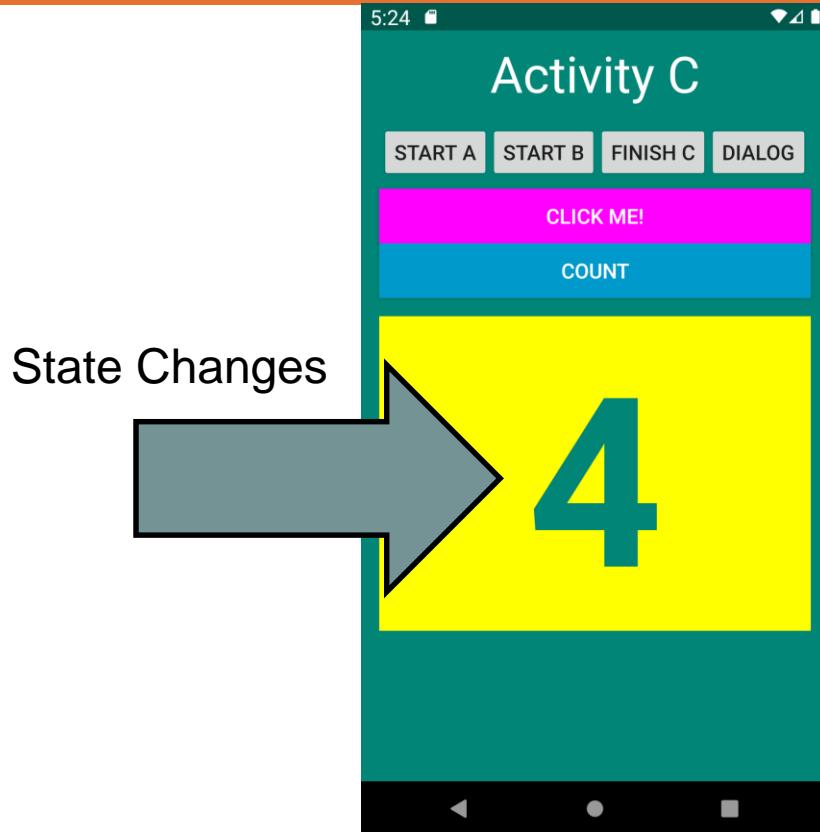
# Launch Activity



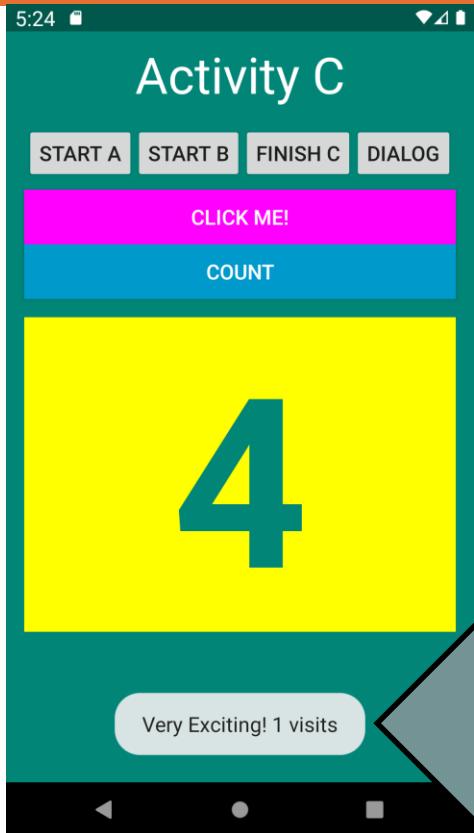
# User Launches Activity C



# Button Click Counts

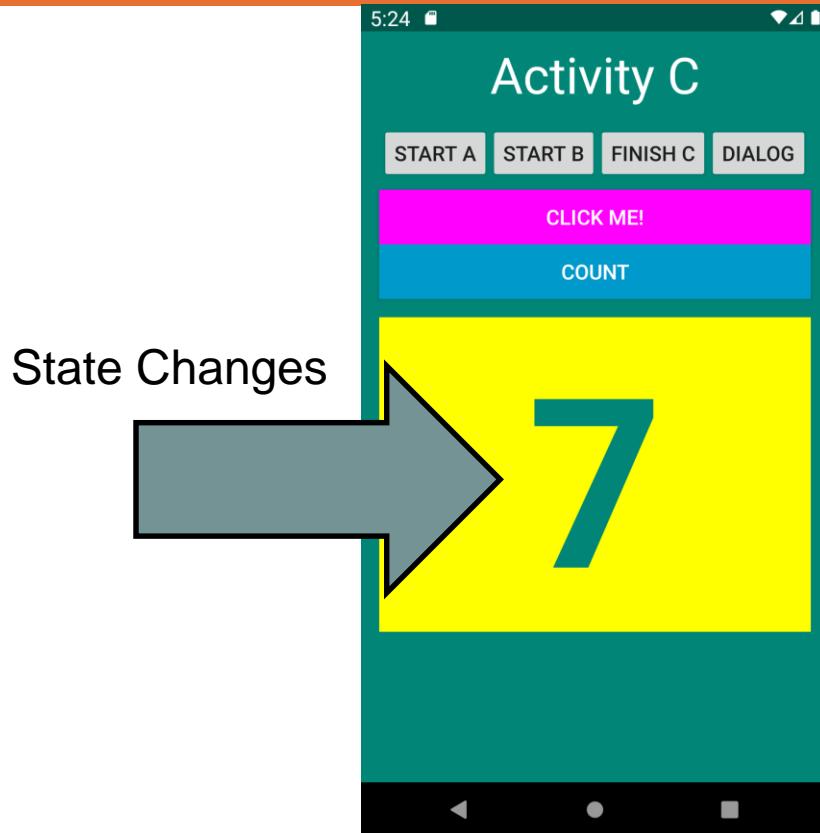


# Screen Visits

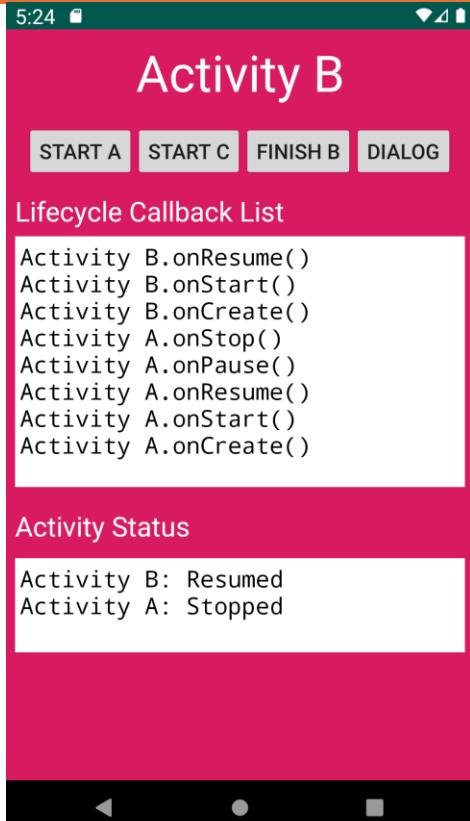


State Changes

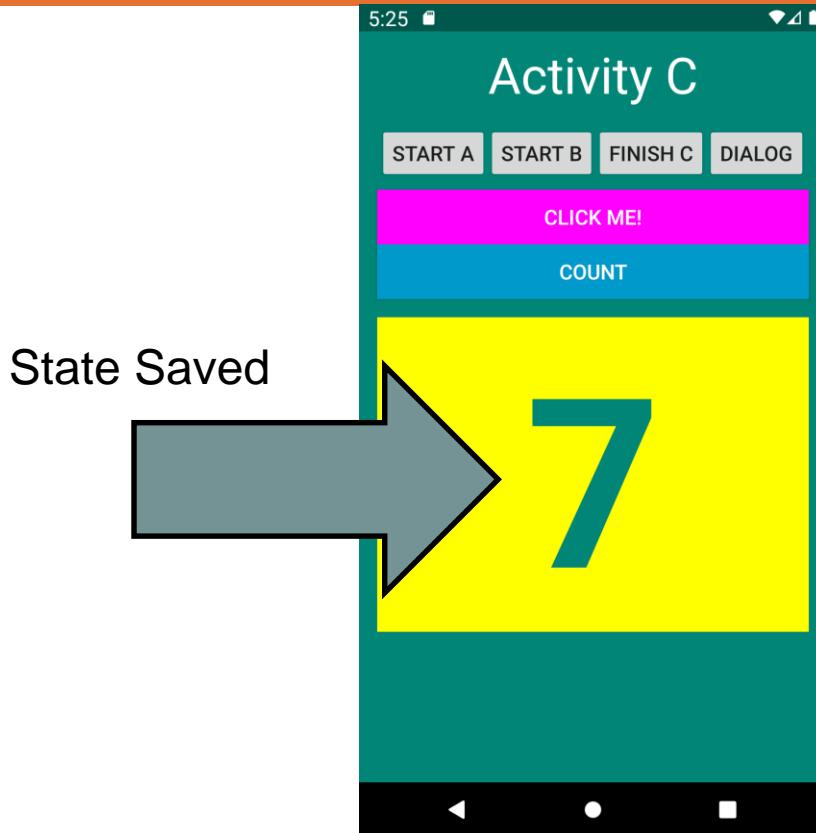
# Button Click Counts



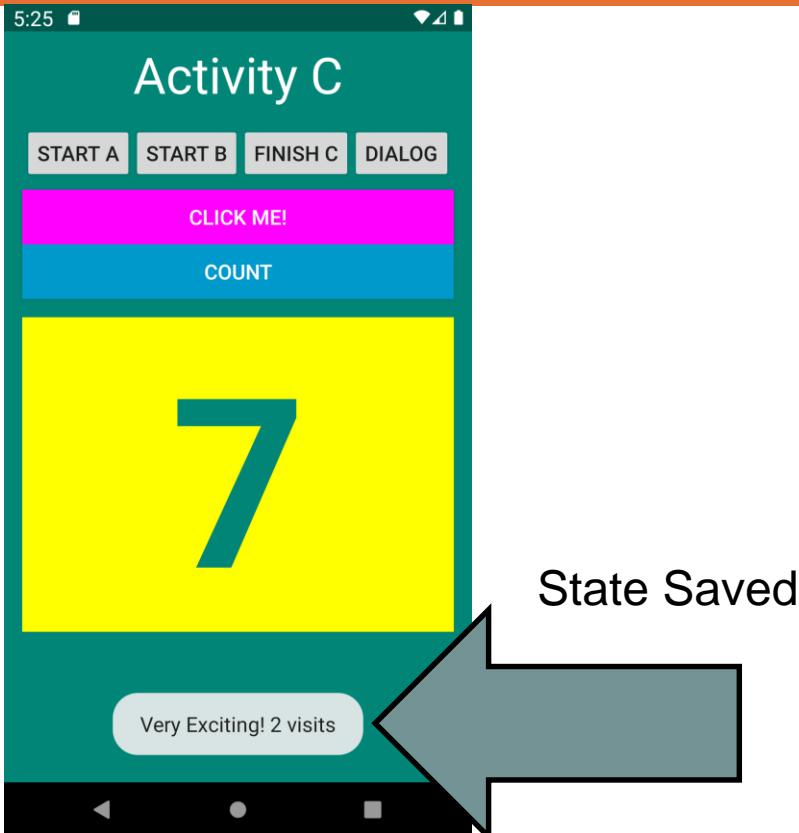
# Launch Activity B



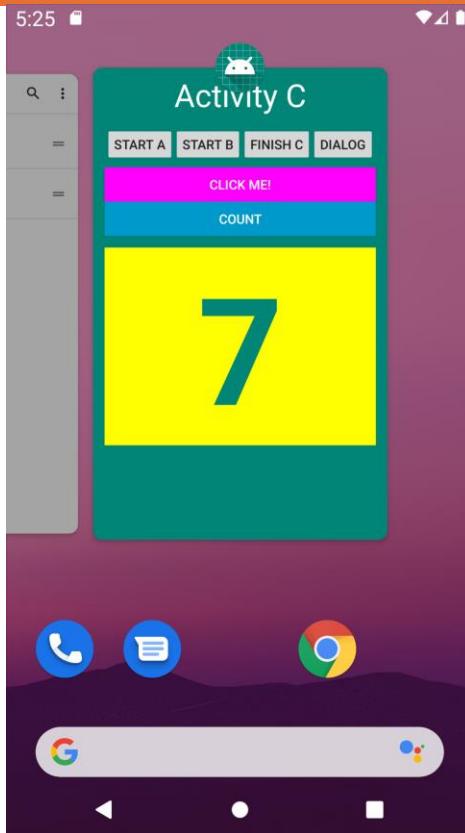
# Click Back Button



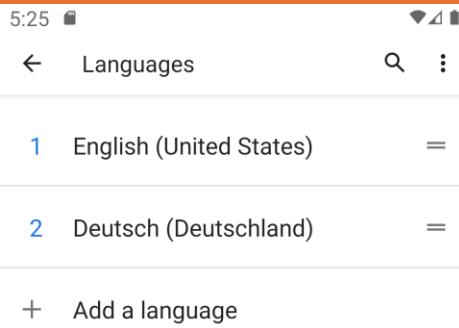
# Screen Visits



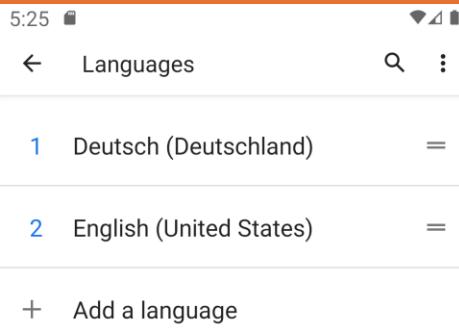
# User Leaves App



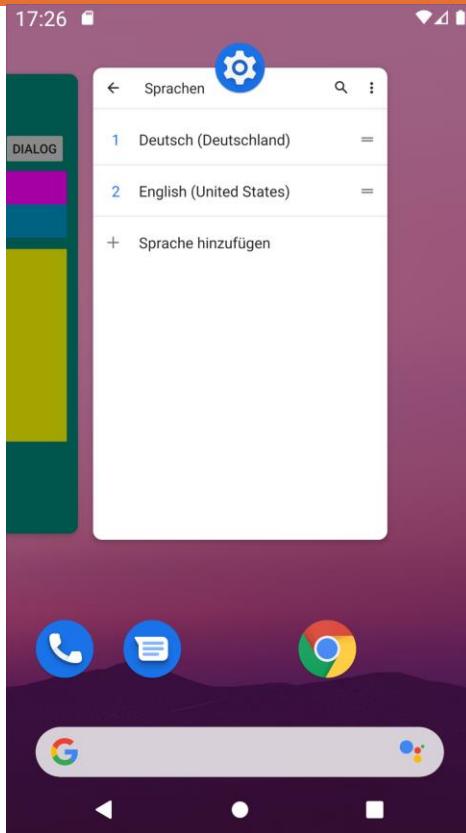
# Configuration Change



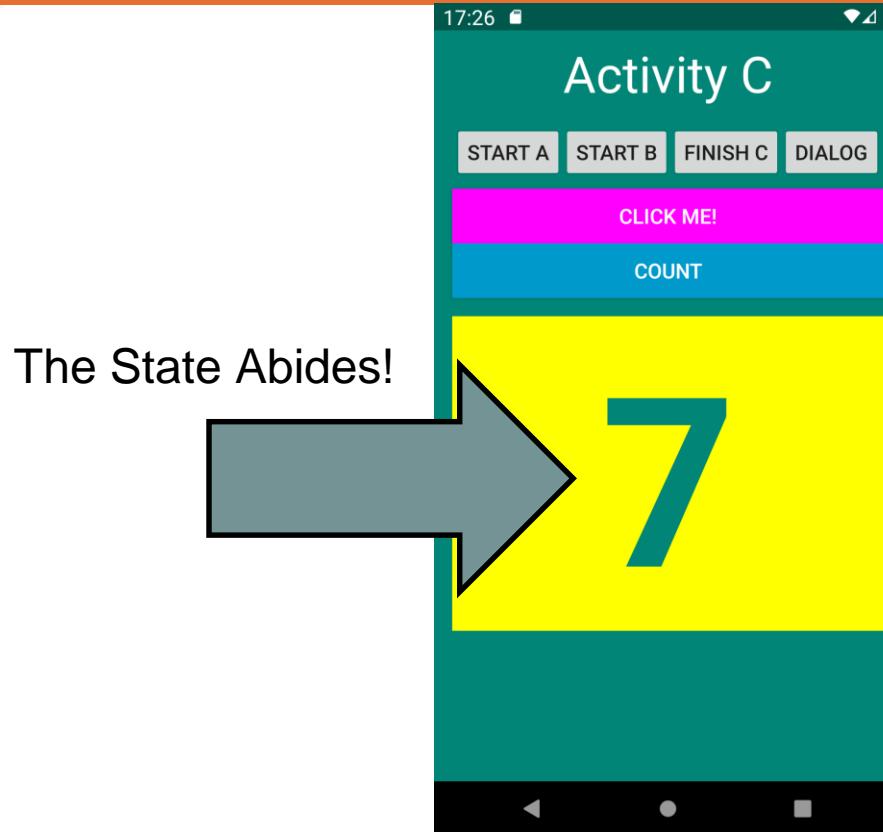
# New Language



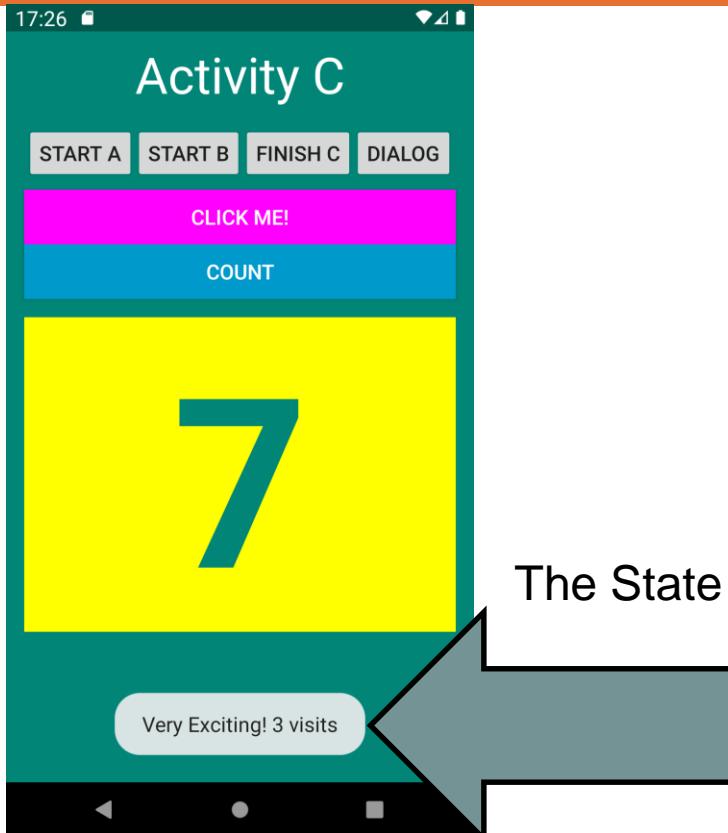
# User Returns to App



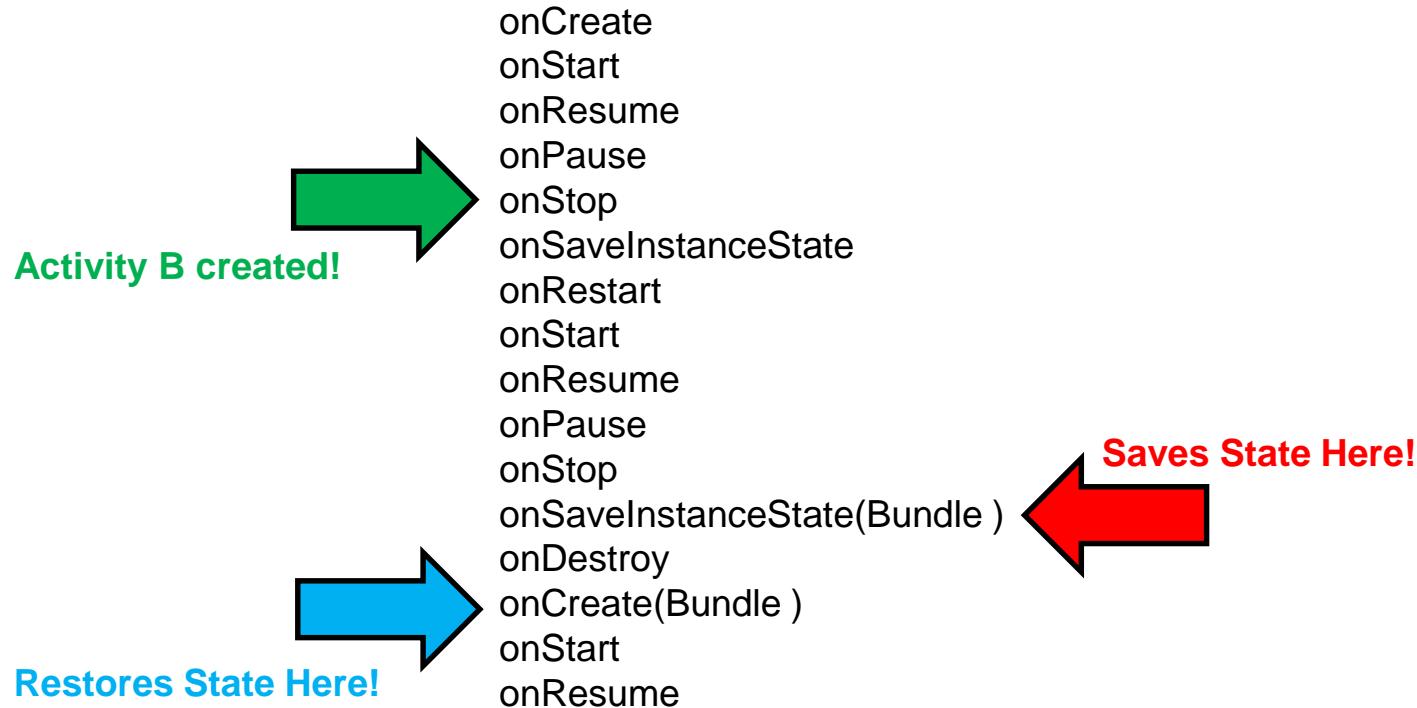
# User Returns to App



# Screen Visits



# Activity C Lifecycle Callbacks



# Mobile Software Development

**CMPU 3036 DT211C/3, DT282/3**

# Today's Lecture

- Threads
- AsyncTask
- Loaders
- AsyncTaskLoader
- Modern Android

# Threads

# MultiTasking

- In 2019 we multitask all the time:
  1. Read and scroll text of Web page while graphics load.
  2. Print document in background while opening another for editing.
  3. Reply to email while new message with large file is quietly downloaded.
- **Threads** make all this convenience possible.

# Recap

- When **OS** starts running a program, it creates a new **process**.
  - A process is a program that is currently executing.
- Every process has **at least one** thread running within it.
- When JVM is started, a new process is created.

# Recap

- You might think of Java code execution as:
  1. starting with main() method
  2. following a path until all statements are completed.
- This is an example of a **single** thread.

# Recap

- 2<sup>nd</sup> thread is always running in JVM i.e. **garbage collection** thread.
- If program includes GUI, JVM starts **more** threads:
  1. One in charge of delivering GUI **events** to methods in program
  2. Another responsible for **painting** GUI window.

# In Java

- Steps to create and run new thread in **Java** are:
  1. **Extend** Java.lang.Thread class.
    - Implement **run** method in this subclass.
  2. Create an **object** of this subclass.
  3. Call **start** method on this object.

# In Java

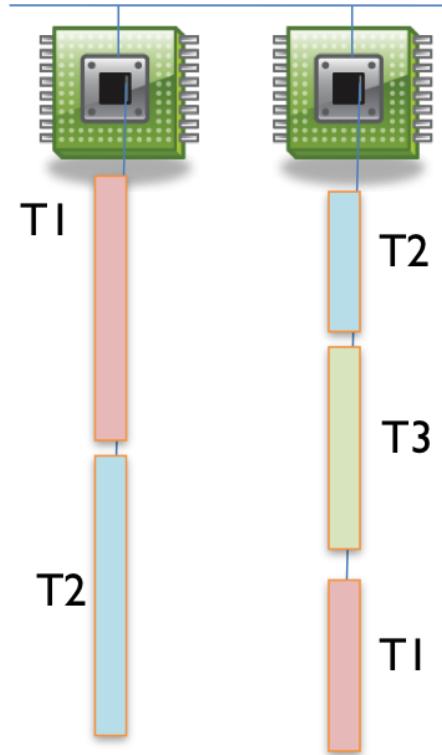
```
public class MyThread extends Thread {  
  
    public void run(){  
        for(int i = 0; i < 100; i++){  
            System.out.println("Thread!");  
        }  
    }  
}
```

# In Java

```
public class MyProgram {  
    public static void main(String[] args){  
        MyThread t = new MyThread();  
        t.start();  
  
        for(int i = 0; i < 100; i++){  
            System.out.println("Main!");  
        }  
    }  
}
```

# In General

- 3 **threads** T1, T2, and T3 running on machine with 2 **processors**.
- First, processor 1 runs T1 and the other T2
- Then processor 2 switches to run T3.
- T2 simply pauses, until its next **time slice** on a processor.





# Back to Android!



- Having a great **idea** is start toward making a cool app.
- Next is focus on app's performance e.g. users want apps that:
  1. Use power **sparingly**.
  2. Start up **quickly**.
  3. **Respond** quickly to user interaction.
- Today we are interested in responsiveness, hence **threads**.

# Single Process

- When an app component (e.g. Activity) starts and **no other** components running:
  - Android starts new **Linux** process with **single** thread of execution.
- By default, all components of app run in the **same** process and thread (called “Main” thread).

# Main Thread

- Independent **path of execution** in a running program
- Code is executed **line by line**.
- App runs on Java thread called "main" or "UI thread"
  1. **Draws** UI on the screen
  2. **Responds** to user actions by handling UI events
- Must be Fast!

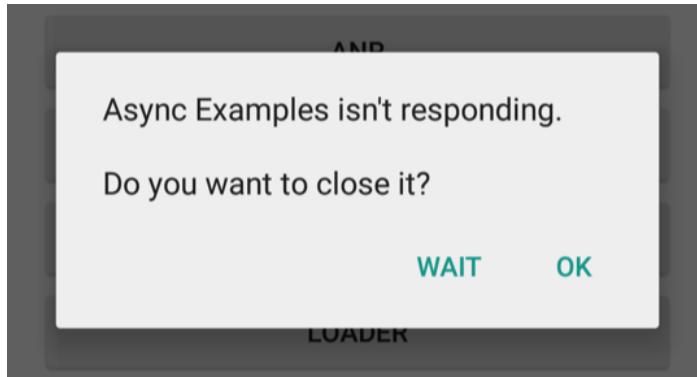
# Main Thread

- Hardware updates screen every 16 milliseconds
- UI thread has 16 ms to do all its work
- If it takes too long, app stutters or **hangs**



# Main Thread

- If the UI waits **too long** for an operation to finish, it becomes unresponsive. After 5 seconds of this...
- The framework shows an Application Not Responding (**ANR**) dialog

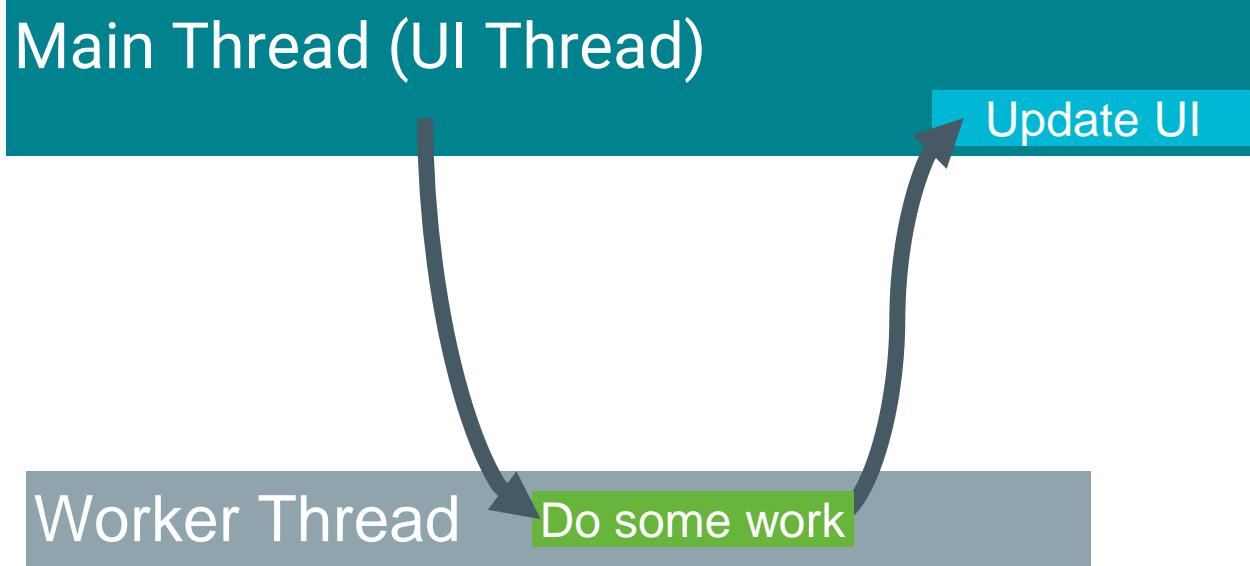


# Long Running Tasks

- Network operations
- Long calculations
- Downloading / uploading files
- Processing images
- Loading data

# Background Threads

- Execute long running tasks on a **background** / worker thread



# 2 Golden Rules

1. Do **not** block the UI thread
  - Complete all work in less than 16 ms for each screen
  - Run **slow** non-UI work on a non-UI thread
2. Do **not** access the Android UI toolkit from outside the UI thread
  - Do UI work **only** on the UI thread

# Worker Threads

- As we know, **vital** not to block UI thread.
- Operations that are **not** instantaneous, should be done in threads.
- However, we can **only** update UI from UI thread.
- Android offers **several** solutions to this e.g. View.post(Runnable)

# Post Runnable

```
public void onClick(View v) {  
    new Thread(new Runnable() {  
        public void run() {  
            final Bitmap bitmap = processBitMap("image.png");  
            mImageView.post(new Runnable() {  
                public void run() {  
                    mImageView.setImageBitmap(bitmap);  
                }});}  
    }).start();  
}
```

# Post Runnable

- Background operation is done from **separate** thread
- **ImageView** is always manipulated from UI thread.
- This kind of code can get complicated and difficult to **Maintain**.
- A better solution is **extending** AsyncTask, which simplifies the whole process.

# AsyncTask

# What is Asynchronous?

## Synchronous

One request at a time



# What is Asynchronous?

## Synchronous

One request at a time



## Asynchronous

Multiple requests at a time

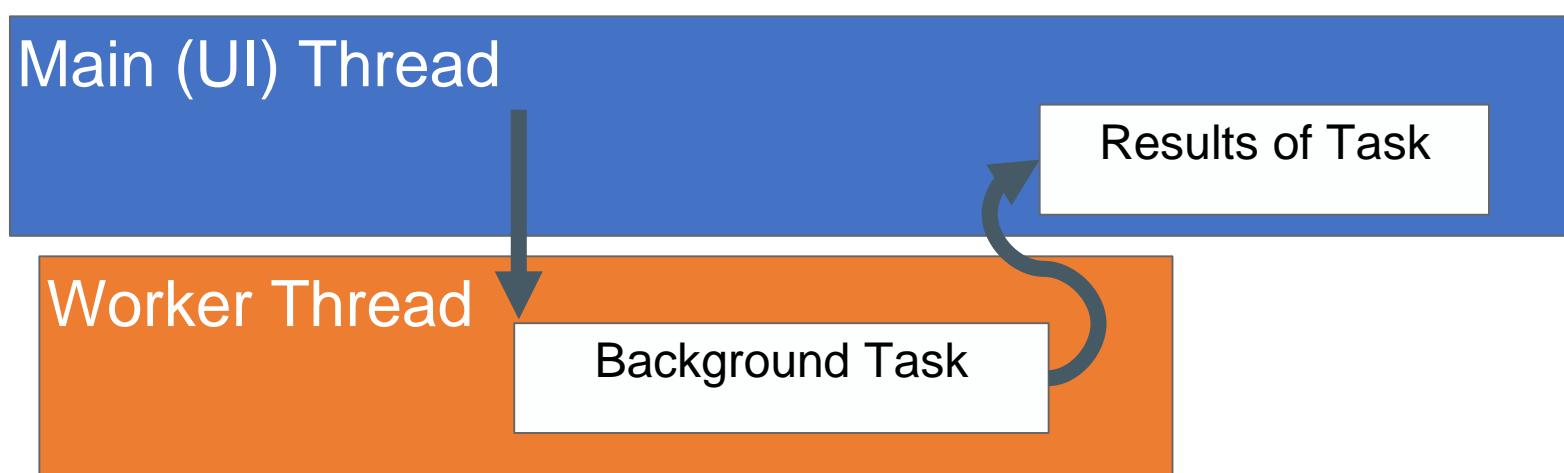


# What is AsyncTask?

- AsyncTask is **simple**, useful way to quickly move work from main thread onto worker threads.
- Performs **blocking** operations in worker thread and publishes results on UI thread.
- Does not require you to handle threads yourself.

# What is AsyncTask?

- Use [AsyncTask](#) to implement **basic** background tasks



# Before Executing

```
protected void onPreExecute() {  
    // display a progress bar  
    // show a toast  
}
```

- Called on the UI thread **before** task executed.
- Used to **set up** task e.g. showing progress bar in UI.

# Background Task

```
protected Bitmap doInBackground(String... query) {  
    // Get the bitmap  
    return bitmap;  
}
```

- Called on **background** thread after onPreExecute() finishes.
- Does background computation that can take long time.
- Params of AsyncTask are passed to it.
- Result of work is **sent** to onPostExecute()

# Report Progress

```
protected void onProgressUpdate(Integer... progress) {  
    setProgressPercent(progress[0]);  
}
```

- Runs on **Main** thread.
- Receives calls from publishProgress() in background thread
- Displays progress on UI **while** background task executing
  - e.g. **animate** progress bar or show logs in text field.

# After Executing

```
protected void onPostExecute(Bitmap result) {  
    // Do something with the bitmap  
}
```

- Called on UI thread **after** the background computation finishes.
  - Process results
  - Publish results to the UI

# Starting Background Task

```
public void loadImage (View view) {  
    String query = mEditText.getText().toString();  
    new MyAsyncTask(query).execute();  
}
```

- To kick off the background task defined in the subclass, call the **execute** method.

# Cancelling a Task

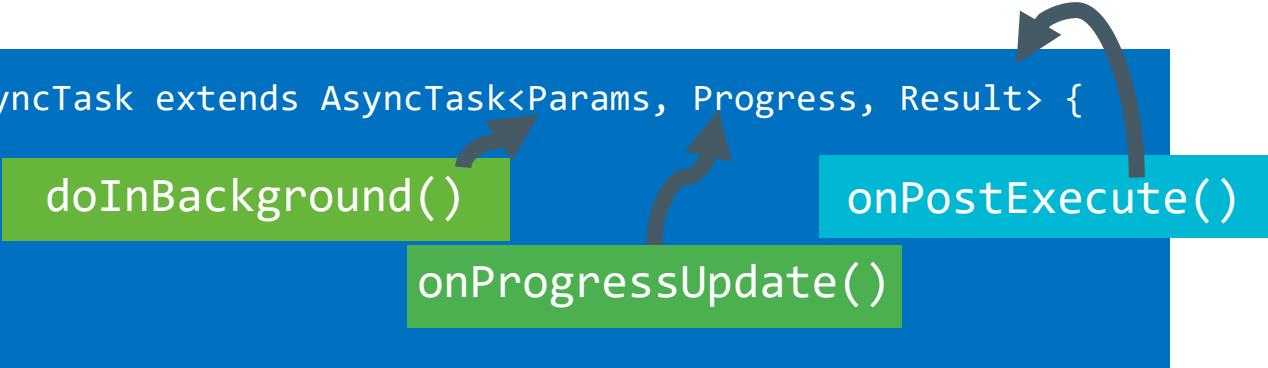
- A task can be **cancelled** by calling cancel.
- Causes isCancelled() to return **true**.
- onCancelled() called **instead** of onPostExecute() after doInBackground() returns.
- To ensure task is cancelled quickly, **check** return value of isCancelled() periodically from doInBackground() e.g. inside a loop.

# Create an AsyncTask

- Subclass AsyncTask
- Include:
  - Types to doInBackground()
  - Type of progress units for onProgressUpdate()
  - Type of result for onPostExecute()

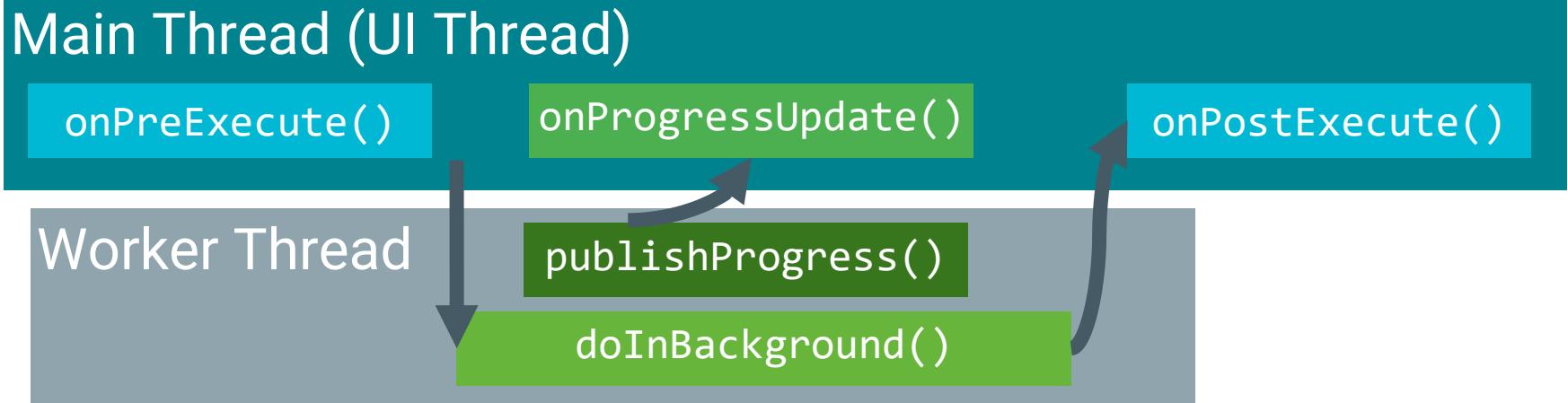
# Create an AsyncTask

```
private class MyAsyncTask extends AsyncTask<Params, Progress, Result> {  
    doInBackground()  
    onProgressUpdate()  
    onPostExecute()  
}  
onProgressUpdate()
```



- **Params** - could be query, URI for filename
- **Progress** - percentage completed, steps done
- **Result** - an image to be displayed
- Use **Void** if no value passed

# AsyncTask In Action



# Example

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {  
}  
}
```

- 1st step is subclass AsyncTask and declare 3 generic types

# Example

```
protected Long doInBackground(URL... urls) {  
    int count = urls.length;  
    long totalSize = 0;  
    for (int i = 0; i < count; i++) {  
        totalSize += Downloader.downloadFile(urls[i]);  
        publishProgress((int) ((i / (float) count) * 100));  
        if (isCancelled()) break;  
    }  
    return totalSize;  
}
```

# Example

```
protected void onProgressUpdate(Integer... progress) {  
    setProgressPercent(progress[0]);  
}
```

```
protected void onPostExecute(Long result) {  
    showDialog("Downloaded " + result + " bytes");  
}
```

# Example

```
new DownloadFilesTask().execute(url1, url2, url3);
```

- Once defined, a task is **executed** very simply.

# AsyncTask Rules

- Task **instance** must be created on UI thread.
- **Execute** must be called on UI thread.
- Do **not** call `onPreExecute`, `onPostExecute`, `doInBackground`,  
`onProgressUpdate` **manually**.
- Task can be executed **only once** (exception thrown if 2<sup>nd</sup> execution  
attempted)

# AsyncTask Limitations

- When device **configuration** changes, Activity is destroyed.
- AsyncTask cannot **connect** to Activity anymore.
- New AsyncTask created for every configuration change.
- Old AsyncTasks stay **around**.
- App may run out of memory or **crash**.

# When to use AsyncTask

- **Short** or interruptible tasks
- Tasks that do **not** need to report back to UI or user
- Lower priority tasks that can be left **unfinished**.
- Use AsyncTaskLoader **otherwise**.

# Loaders

# What is a Loader?

- Provides **asynchronous** loading of data.
- **Reconnects** to Activity after configuration change.
- Can **monitor** changes in data source and deliver new data.
- Callbacks implemented in Activity.
- Many **types** of loaders available
  - [AsyncTaskLoader](#), [CursorLoader](#)

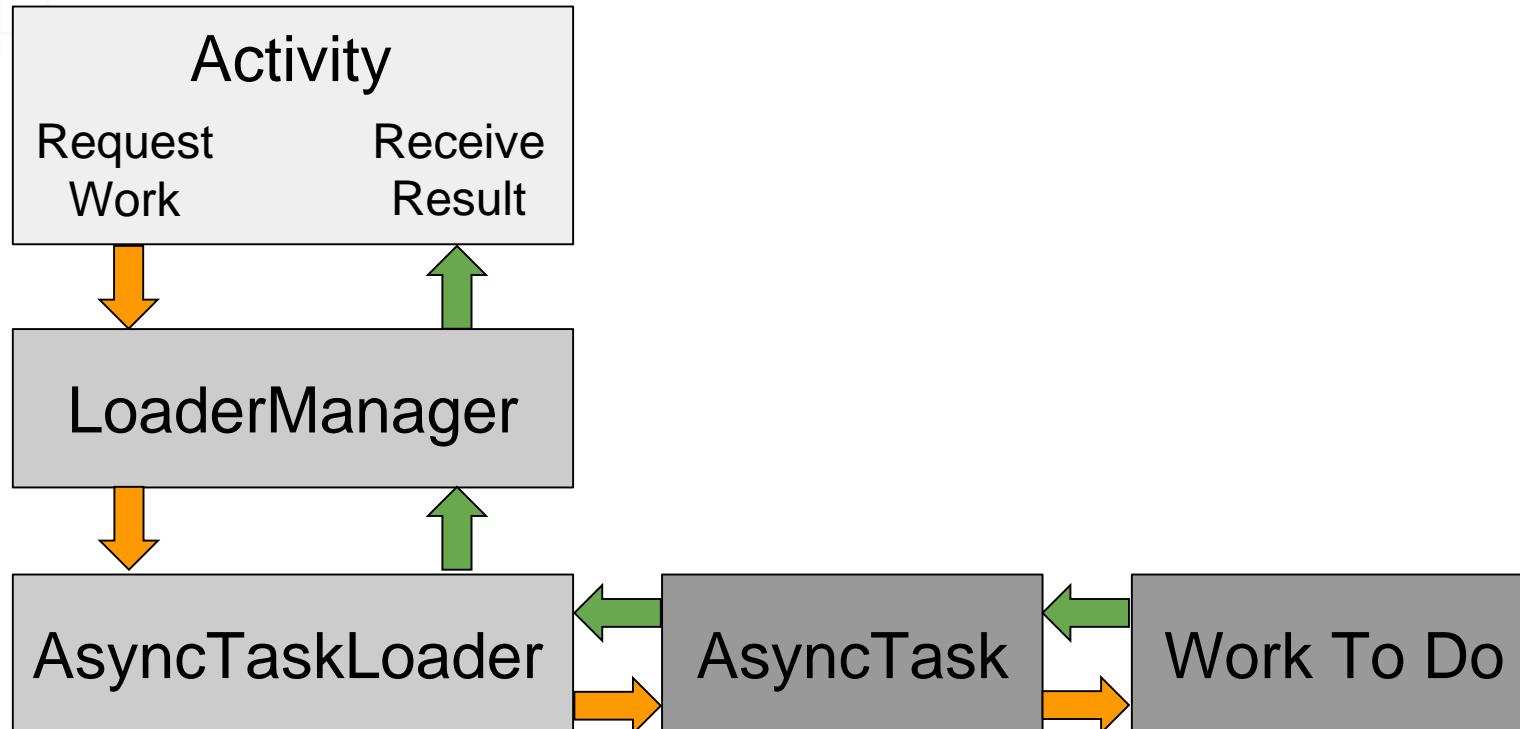
# Why Use Loaders?

- Execute tasks OFF the UI thread.
- Loader **Manager** handles configuration changes for you.
- Efficiently implemented by the framework.
- Users don't have to **wait** for data to load.

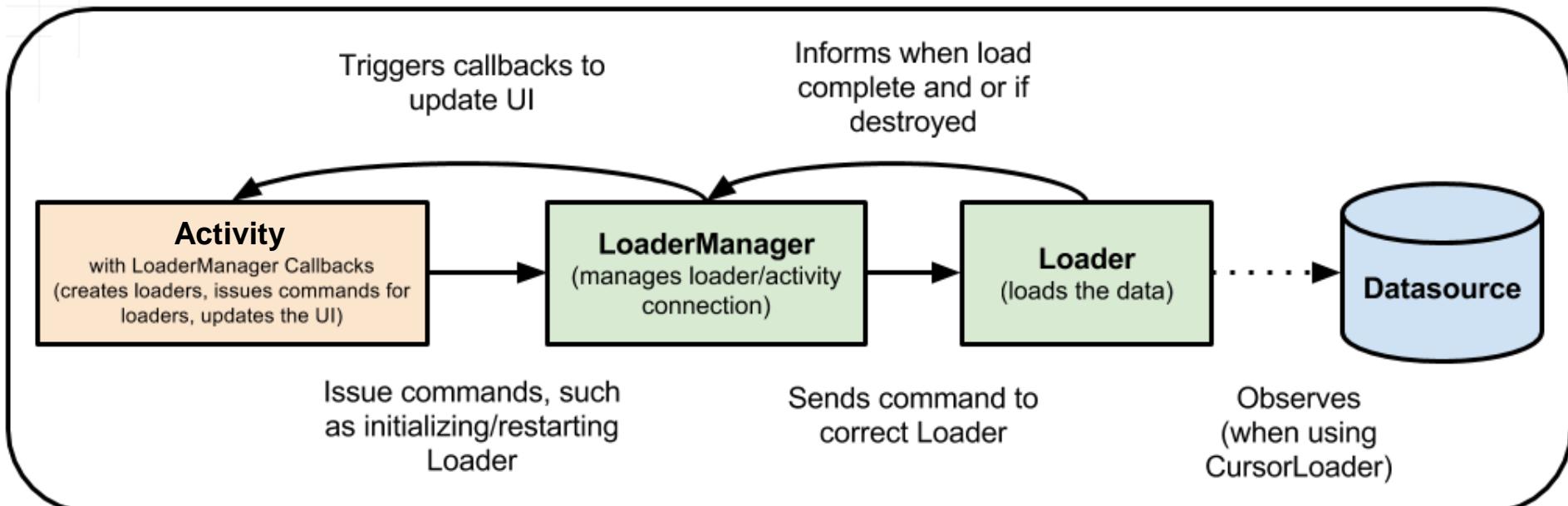
# What is a Loader Manager?

- Manages Loader functions via callbacks
- Can manage **multiple** loaders
  - loader for **database** data, for AsyncTask data, for internet data etc.
- Creates and starts a loader, or reuses an existing one, including its data.

# AsyncTask Loader



# Cursor Loader

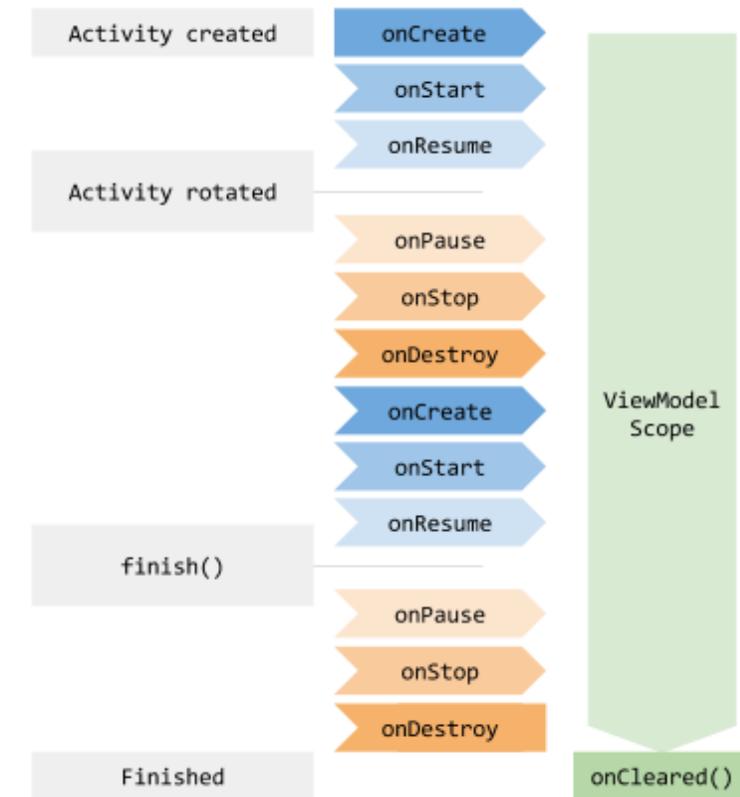


# Modern Android

# Replacing Loaders

- Loaders like CursorLoader keep UI data in sync with a database.
  - e.g. when a value in database changes, CursorLoader triggers a reload of data and updates UI.
- **ViewModel** with a few other classes is a modern approach to replacing the Loader.
- ViewModel ensures that the data **survives** a configuration change.

# ViewModel Lifecycle



# ViewModel Lifecycle

- If the Android System **destroys** and **re-creates** an Activity, for simple data onSaveInstanceState can be used.
- This approach is suitable only for small amounts of data.
- The ViewModel approach can be used to restore more complex data like **state** of a scrollable list of items.

# Mobile Software Development

**CMPU 3036 DT211C/3, DT282/3**



# Today's Lecture

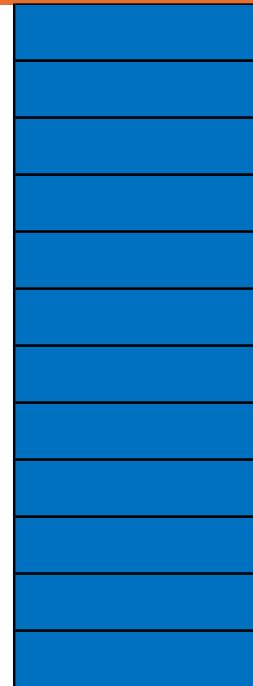
- RecyclerView Components
- Implementing a RecyclerView

# What is a RecyclerView?

- RecyclerView is **scrollable** container for large data sets.

# What is a RecyclerView?

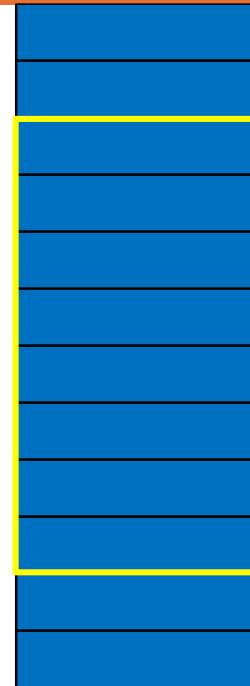
- RecyclerView is **scrollable** container for large data sets.



Large data set

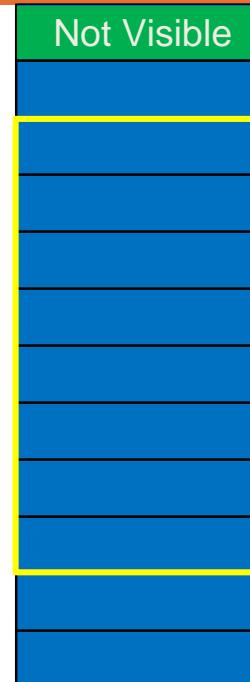
# What is a RecyclerView?

- RecyclerView is **scrollable** container for large data sets.



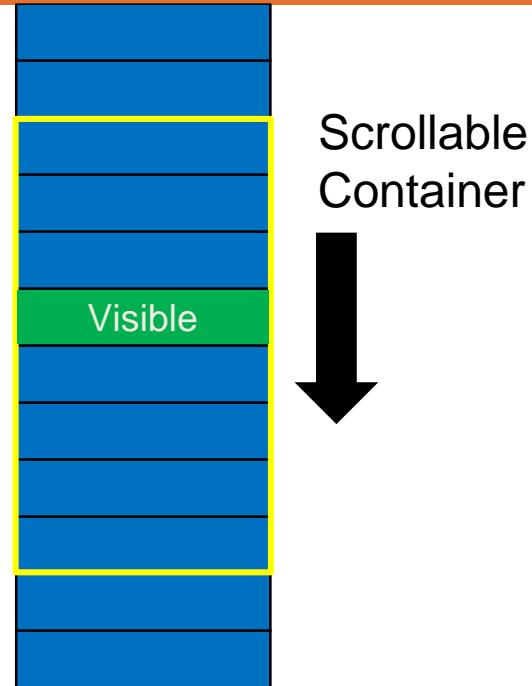
# What is a RecyclerView?

- RecyclerView is **scrollable** container for large data sets.



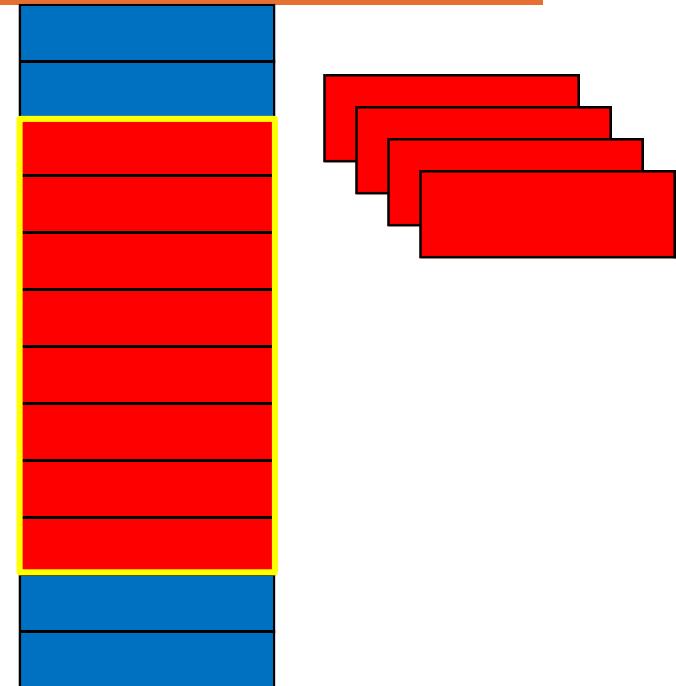
# What is a RecyclerView?

- RecyclerView is **scrollable** container for large data sets.



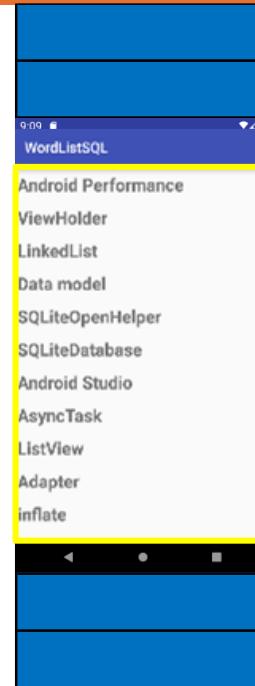
# What is a RecyclerView?

- RecyclerView is **scrollable** container for large data sets.
- Efficient
  - Uses and reuses **limited number** of View elements



# What is a RecyclerView?

- RecyclerView is **scrollable** container for large data sets.
- Efficient
  - Uses and reuses limited number of View elements
  - Updates changing data **fast**



# RecyclerView Components

# Overview

## 1. Data

# Overview

1. Data
2. [RecyclerView](#) scrolling list for list items

# Overview

1. **Data**
2. [RecyclerView](#) scrolling list for list items
3. **Layout** for one item of data - XML file

# Overview

1. **Data**
2. [RecyclerView](#) scrolling list for list items
3. **Layout** for one item of data - XML file
4. [Layout Manager](#) handles the organisation of UI components in a View.

# Overview

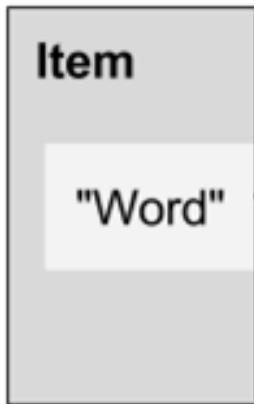
1. **Data**
2. RecyclerView scrolling list for list items
3. **Layout** for one item of data - XML file
4. Layout Manager handles the organisation of UI components in a View.
5. Adapter connects data to the RecyclerView.

# Overview

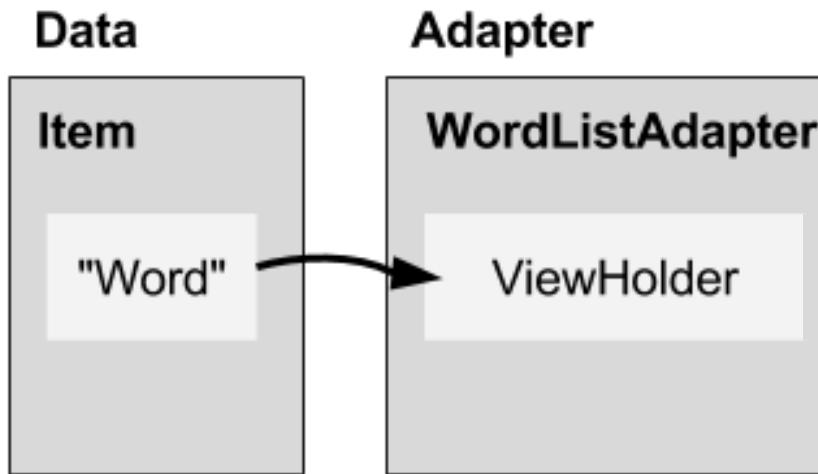
1. **Data**
2. RecyclerView scrolling list for list items
3. **Layout** for one item of data - XML file
4. Layout Manager handles the organisation of UI components in a View.
5. Adapter connects data to the RecyclerView.
6. ViewHolder has view information for displaying one item.

# How Components Fit Together

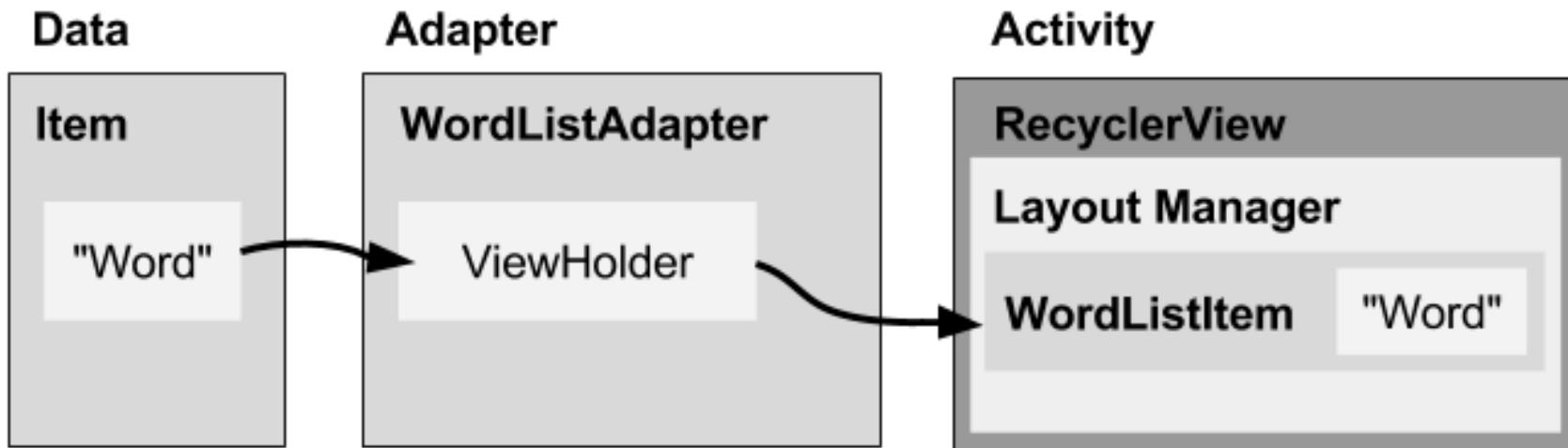
Data



# How Components Fit Together

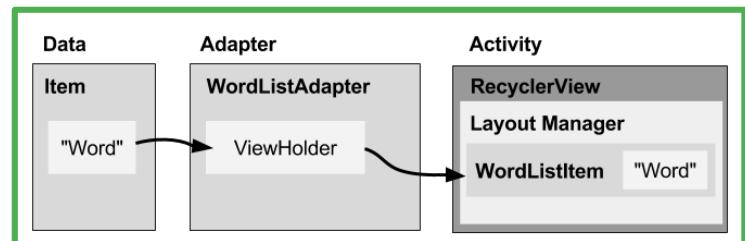


# How Components Fit Together



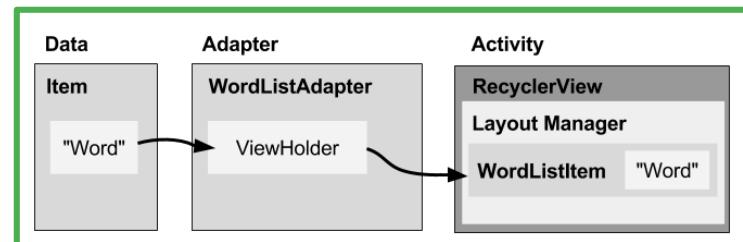
# What is a Layout Manager?

- Each ViewGroup has a Layout Manager



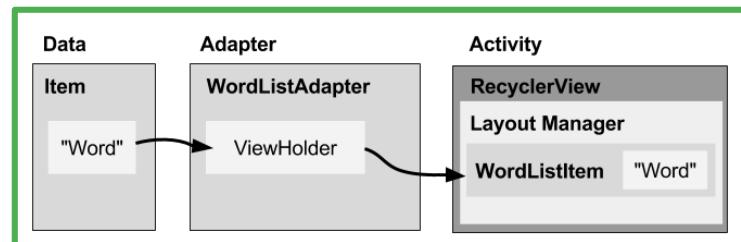
# What is a Layout Manager?

- Each ViewGroup has a Layout Manager
- Used to position View items inside a RecyclerView



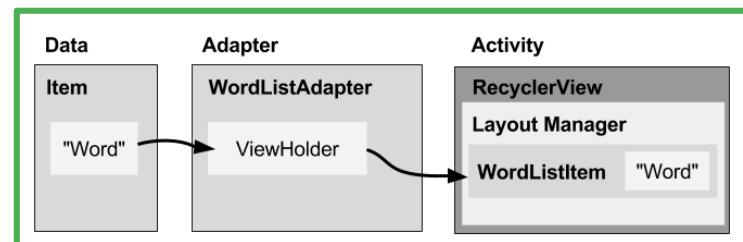
# What is a Layout Manager?

- Each ViewGroup has a Layout Manager
- Used to position View items inside a RecyclerView
- Reuses View items that are **no longer visible** to the user



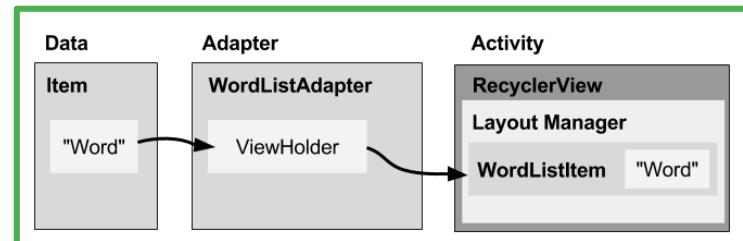
# What is a Layout Manager?

- Each ViewGroup has a Layout Manager
- Used to position View items inside a RecyclerView
- Reuses View items that are **no longer visible** to the user
- Built-in layout managers
  - LinearLayoutManager
  - GridLayoutManager
  - StaggeredGridLayoutManager



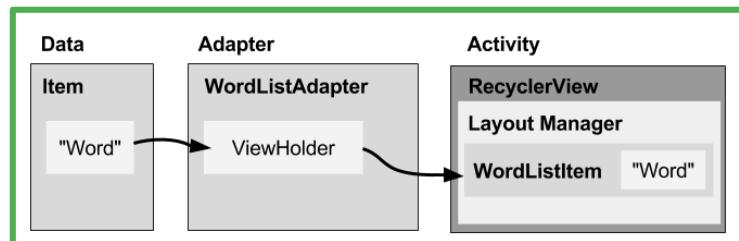
# What is a Layout Manager?

- Each ViewGroup has a Layout Manager
- Used to position View items inside a RecyclerView
- Reuses View items that are **no longer visible** to the user
- Built-in layout managers
  - LinearLayoutManager
  - GridLayoutManager
  - StaggeredGridLayoutManager
- Extend RecyclerView.LayoutManager



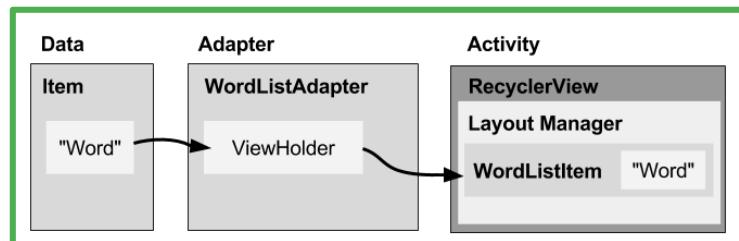
# What is an Adapter?

- Helps **incompatible** interfaces work together



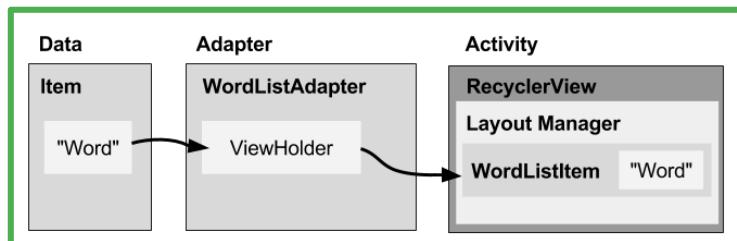
# What is an Adapter?

- Helps **incompatible** interfaces work together
  - Example: Takes data from database Cursor and prepares strings to put into a View



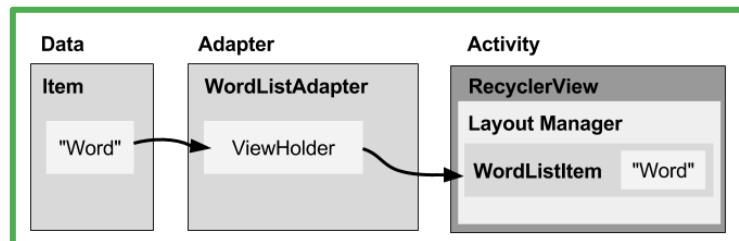
# What is an Adapter?

- Helps **incompatible** interfaces work together
  - Example: Takes data from database Cursor and prepares strings to put into a View
- **Intermediary** between data and View



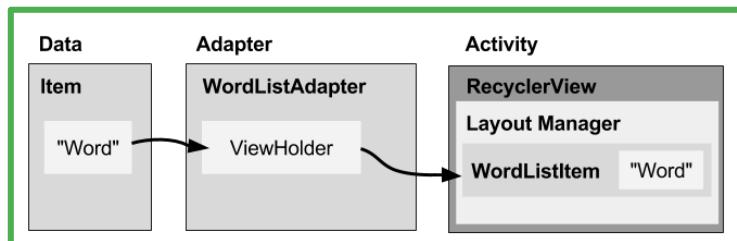
# What is an Adapter?

- Helps **incompatible** interfaces work together
  - Example: Takes data from database Cursor and prepares strings to put into a View
- **Intermediary** between data and View
- **Manages** creating, updating, adding, deleting View items as underlying data changes.



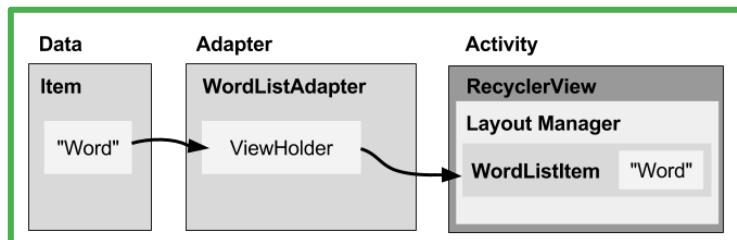
# What is a ViewHolder?

- Used by the **Adapter** to prepare one View with data for one list item



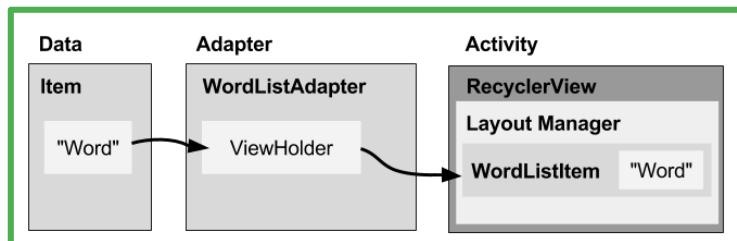
# What is a ViewHolder?

- Used by the **Adapter** to prepare one View with data for one list item
- **Layout** specified in an XML resource file



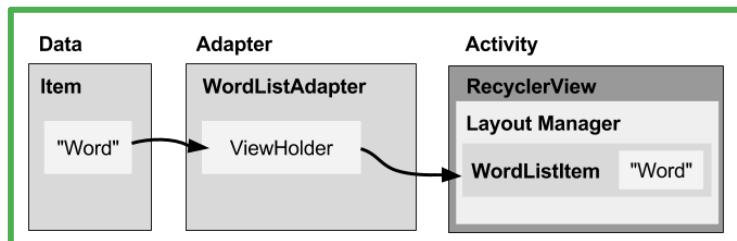
# What is a ViewHolder?

- Used by the **Adapter** to prepare one View with data for one list item
- **Layout** specified in an XML resource file
- Can have **clickable** elements



# What is a ViewHolder?

- Used by the **Adapter** to prepare one View with data for one list item
- **Layout** specified in an XML resource file
- Can have **clickable** elements
- Is **placed** by the Layout Manager



# Implementing RecyclerView

# Steps Summary

1. Add RecyclerView dependency to build.gradle if needed

# Steps Summary

1. Add RecyclerView dependency to build.gradle if needed
2. Add RecyclerView to **layout**

# Steps Summary

1. Add RecyclerView dependency to build.gradle if needed
2. Add RecyclerView to **layout**
3. Create XML layout for **item**

# Steps Summary

1. Add RecyclerView dependency to build.gradle if needed
2. Add RecyclerView to **layout**
3. Create XML layout for **item**
4. **Extend** RecyclerView.Adapter

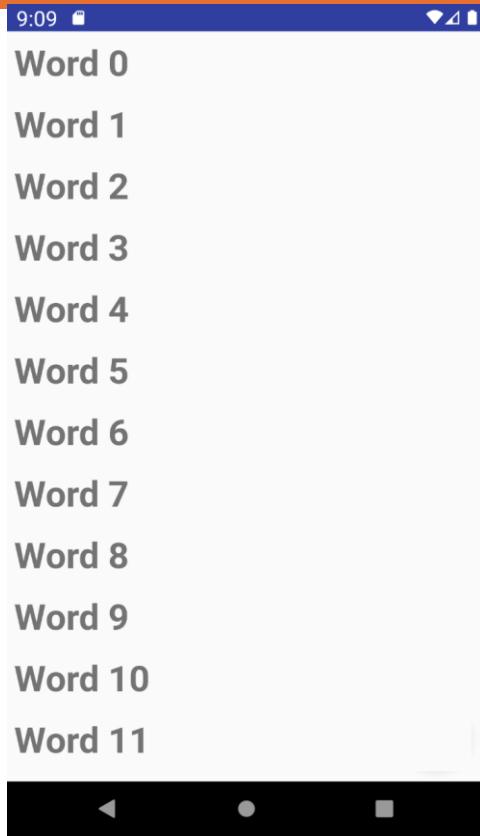
# Steps Summary

1. Add RecyclerView dependency to build.gradle if needed
2. Add RecyclerView to **layout**
3. Create XML layout for **item**
4. **Extend** RecyclerView.Adapter
5. **Extend** RecyclerView.ViewHolder

# Steps Summary

1. Add RecyclerView dependency to build.gradle if needed
2. Add RecyclerView to **layout**
3. Create XML layout for **item**
4. **Extend** RecyclerView.Adapter
5. **Extend** RecyclerView.ViewHolder
6. In Activity onCreate(), create RecyclerView **with** Adapter and Layout Manager

# Example



# Adapter

WordListAdapter  
extends  
Adapter

WordListAdapter  
**extends**  
Adapter

```
LinkedList<String> mWordList
```

WordListAdapter  
**extends**  
Adapter

```
LinkedList<String> mWordList
```

```
int getItemCount()
```

WordListAdapter  
extends  
Adapter

WordViewHolder  
extends  
ViewHolder

WordListAdapter  
extends  
Adapter

WordViewHolder  
extends  
ViewHolder

    TextView wordItemView

WordListAdapter  
extends  
Adapter

WordViewHolder  
extends  
ViewHolder

```
WordViewHolder(View itemView) {  
}  
}
```

WordListAdapter  
extends  
Adapter

```
WordViewHolder onCreateViewHolder(      ) {  
    ...  
}
```

WordViewHolder  
extends  
ViewHolder

```
WordViewHolder(View itemView) {  
    ...  
}
```

WordListAdapter  
extends  
Adapter

```
LayoutInflater mInflater  
  
WordViewHolder onCreateViewHolder(      ) {  
  
}
```

WordViewHolder  
extends  
ViewHolder

```
WordViewHolder(View itemView) {  
  
}
```

WordListAdapter  
extends  
Adapter

```
LayoutInflater mInflater  
  
WordViewHolder onCreateViewHolder(      ) {  
}  
}
```

WordViewHolder  
extends  
ViewHolder

```
WordViewHolder(View itemView) {  
}  
}
```

wordlist\_item.xml

```
<LinearLayout>  
  <TextView  
    id="@+id/word" />  
</LinearLayout>
```

WordListAdapter  
extends  
Adapter

```
LayoutInflater mInflater

WordViewHolder onCreateViewHolder(      ){
    View mItemView = mInflater.inflate(wordlist_item)
}
```

WordViewHolder  
extends  
ViewHolder

```
WordViewHolder(View itemView) {
}
```

wordlist\_item.xml

```
<LinearLayout      >
    <TextView
        id="@+id/word" />
</LinearLayout>
```

WordListAdapter  
extends  
Adapter

```
LayoutInflater mInflater

WordViewHolder onCreateViewHolder(      ){
    View mItemView = mInflater.inflate(wordlist_item)
}
```

WordViewHolder  
extends  
ViewHolder

```
WordViewHolder(View itemView) {
    wordItemView = itemView.findViewById(word)
}
```

wordlist\_item.xml

```
<LinearLayout      >
    <TextView
        id="@+id/word" />
</LinearLayout>
```

WordListAdapter  
extends  
Adapter

```
LayoutInflater mInflater  
  
WordViewHolder onCreateViewHolder(      ) {  
    View mItemView = mInflater.inflate(wordlist_item)  
}
```

wordlist\_item.xml

```
<LinearLayout      >  
    <TextView  
        id="@+id/word" />  
</LinearLayout>
```

WordViewHolder  
extends  
ViewHolder

```
WordViewHolder(View itemView) {  
    wordItemView = itemView.findViewById(word)  
}
```

WordListAdapter  
extends  
Adapter

```
LayoutInflater mInflater

WordViewHolder onCreateViewHolder(      ){
    View mItemView = mInflater.inflate(wordlist_item)
    return new WordViewHolder(mItemView)
}
```

WordViewHolder  
extends  
ViewHolder

```
WordViewHolder(View itemView) {
    wordItemView = itemView.findViewById(word)
}
```

wordlist\_item.xml

```
<LinearLayout      >
    <TextView
        id="@+id/word" />
</LinearLayout>
```

WordListAdapter  
extends  
Adapter

```
onBindViewHolder(WordViewHolder holder, int position){  
}  
}
```

WordViewHolder  
extends  
ViewHolder

```
TextView wordItemView
```

WordListAdapter  
**extends**  
Adapter

```
LinkedList<String> mWordList

onBindViewHolder(WordViewHolder holder, int position){
    String mCurrent = mWordList.get(position)

}
```

WordViewHolder  
**extends**  
ViewHolder

WordListAdapter  
extends  
Adapter

```
onBindViewHolder(WordViewHolder holder, int position){  
  
    holder.wordItemView.setText(mCurrent)  
}
```

WordViewHolder  
extends  
ViewHolder

```
TextView wordItemView
```

# RecyclerView

RecyclerView

Adapter mAdapter

LayoutManager mLayout

Recycler mRecyclerView

RecyclerView

Adapter mAdapter

LayoutManager mLayout

Recycler mRecyclerView

setAdapter(Adapter )

setLayoutManager(LayoutManager )

RecyclerView

```
Adapter mAdapter  
LayoutManager mLayout  
Recycler mRecyclerView  
  
setAdapter(Adapter )  
setLayoutManager(LayoutManager )  
  
Adapter getAdapter()
```

# Activity

## MainActivity

```
onCreate(      ){
    setContentView(activity_main)
}

}
```

## activity\_main.xml

```
<include
layout="@layout/content_main"/>
```

## MainActivity

```
RecyclerView mRecyclerView
```

```
onCreate(    ){
    setContentView(activity_main)
}
```

## activity\_main.xml

```
<include
    layout="@layout/content_main"/>
```

## content\_main.xml

```
<RecyclerView
    id="@+id/recyclerview"
/>
```

## MainActivity

```
RecyclerView mRecyclerView  
  
onCreate(    ){  
    setContentView(activity_main)  
    mRecyclerView = findViewById(recyclerview)  
  
}  
}
```

## activity\_main.xml

```
<include  
layout="@layout/content_main"/>
```

## content\_main.xml

```
<RecyclerView  
    id="@+id/recyclerview"  
/>
```

## MainActivity

```
RecyclerView mRecyclerView
```

```
onCreate(    ){
    setContentView(activity_main)
    mRecyclerView = findViewById(recyclerview)
}

}
```

## activity\_main.xml

```
<include  
layout="@layout/content_main"/>
```

## content\_main.xml

```
<RecyclerView  
    id="@+id/recyclerview"  
</RecyclerView>
```

## MainActivity

```
onCreate(      ){  
}  
}
```

## MainActivity

```
onCreate(      ){  
  
    for (int i = 0; i < 20; i++) {  
  
    }  
  
}
```

19

18

17

16

15

14

13

12

11

10

9

8

7

6

5

4

3

2

1

0

## MainActivity

```
LinkedList<String> mWordList = new LinkedList<>()

onCreate(      ){
    for (int i = 0; i < 20; i++) {
        mWordList.addLast("Word " + i);
    }
}
```

19	Word 19
18	Word 18
17	Word 17
16	Word 16
15	Word 15
14	Word 14
13	Word 13
12	Word 12
11	Word 11
10	Word 10
9	Word 9
8	Word 8
7	Word 7
6	Word 6
5	Word 5
4	Word 4
3	Word 3
2	Word 2
1	Word 1
0	Word 0

## MainActivity

```
WordListAdapter mWordListAdapter

onCreate(    ){
    mWordListAdapter = new WordListAdapter(mWordList)
}

}
```

19	Word 19
18	Word 18
17	Word 17
16	Word 16
15	Word 15
14	Word 14
13	Word 13
12	Word 12
11	Word 11
10	Word 10
9	Word 9
8	Word 8
7	Word 7
6	Word 6
5	Word 5
4	Word 4
3	Word 3
2	Word 2
1	Word 1
0	Word 0

WordListAdapter  
extends  
Adapter

LinkedList<String> mWordList

19	Word 19
18	Word 18
17	Word 17
16	Word 16
15	Word 15
14	Word 14
13	Word 13
12	Word 12
11	Word 11
10	Word 10
9	Word 9
8	Word 8
7	Word 7
6	Word 6
5	Word 5
4	Word 4
3	Word 3
2	Word 2
1	Word 1
0	Word 0

## MainActivity

```
RecyclerView mRecyclerView  
  
onCreate(      ){  
  
    mRecyclerView.setAdapter(mWordListAdapter)  
    LinearLayoutManager lm = new LinearLayoutManager()  
    mRecyclerView.setLayoutManager(lm)  
  
}  
}
```

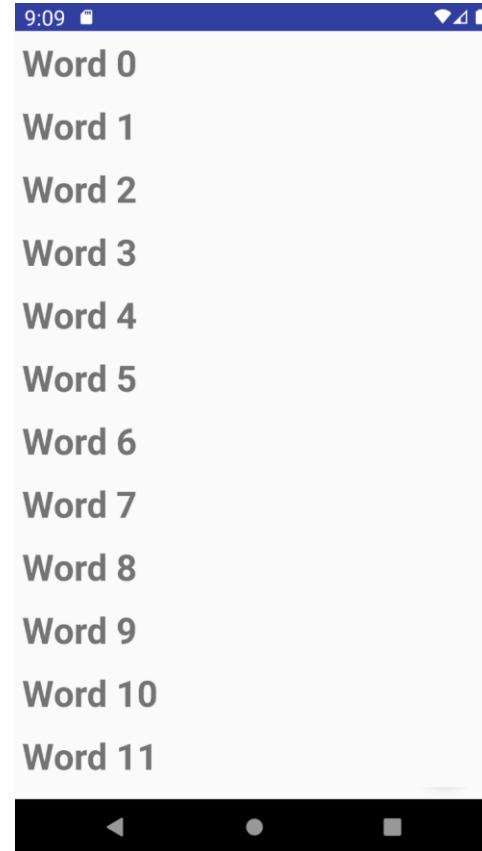
## Data Adapter

19	Word 19
18	Word 18
17	Word 17
16	Word 16
15	Word 15
14	Word 14
13	Word 13
12	Word 12
11	Word 11
10	Word 10
9	Word 9
8	Word 8
7	Word 7
6	Word 6
5	Word 5
4	Word 4
3	Word 3
2	Word 2
1	Word 1
0	Word 0

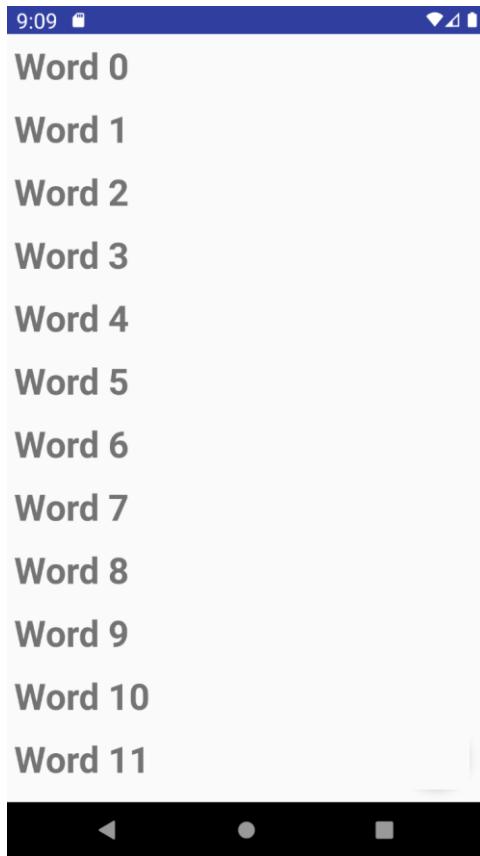
## Recycler

11	ViewHolder
10	ViewHolder
9	ViewHolder
8	ViewHolder
7	ViewHolder
6	ViewHolder
5	ViewHolder
4	ViewHolder
3	ViewHolder
2	ViewHolder
1	ViewHolder
0	ViewHolder

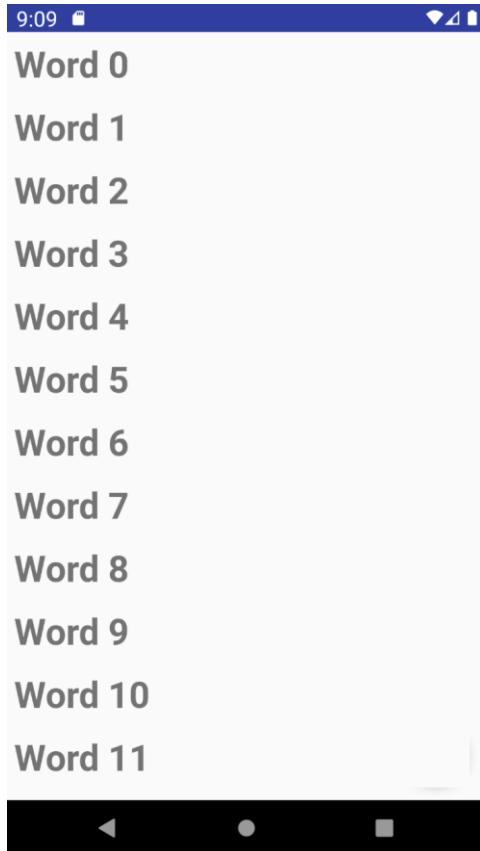
## LinearLayoutManager



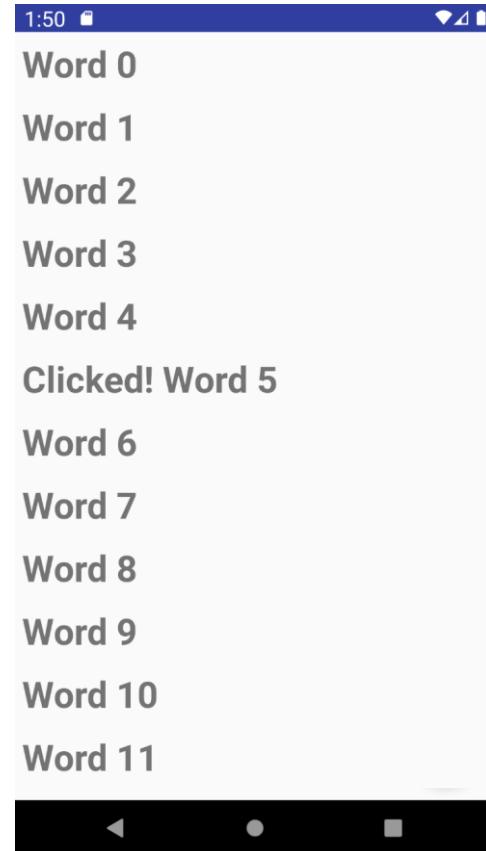
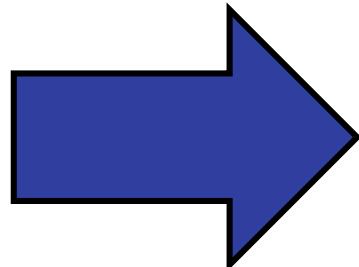
# Add Interactivity



User clicks on item  
in list e.g. Word 5



User clicks on item  
in list e.g. Word 5



```
WordListAdapter  
extends  
Adapter
```

```
WordViewHolder  
extends  
ViewHolder  
implements  
OnClickListener
```

WordListAdapter  
extends  
Adapter

WordViewHolder  
extends  
ViewHolder  
implements  
OnClickListener

```
onClick(View view) {  
  
}
```

WordListAdapter  
extends  
Adapter

WordViewHolder  
extends  
ViewHolder  
implements  
OnClickListener

```
onClick(View view) {  
    int mPosition = getLayoutPosition()  
}
```

WordListAdapter  
extends  
Adapter

WordViewHolder  
extends  
ViewHolder  
implements  
OnClickListener

```
onClick(View view) {  
    int mPosition = getLayoutPosition()  
    String element = mWordList.get(mPosition)  
}
```

7	Word 7
6	Word 6
5	Word 5
4	Word 4
3	Word 3
2	Word 2
1	Word 1
0	Word 0

WordListAdapter  
extends  
Adapter

```
LinkedList<String> mWordList
```

WordViewHolder  
extends  
ViewHolder  
implements  
OnClickListener

```
onClick(View view) {  
    int mPosition = getLayoutPosition()  
    String element = mWordList.get(mPosition)  
    mWordList.set(mPosition, "Clicked! " + element)  
}
```

7	Word 7
6	Word 6
5	Clicked! Word 5
4	Word 4
3	Word 3
2	Word 2
1	Word 1
0	Word 0

WordListAdapter  
extends  
Adapter

```
LinkedList<String> mWordList
```

WordViewHolder  
extends  
ViewHolder  
implements  
OnClickListener

```
onClick(View view) {  
    int mPosition = getLayoutPosition()  
    String element = mWordList.get(mPosition)  
    mWordList.set(mPosition, "Clicked! " + element)  
    mAdapter.notifyDataSetChanged()  
}
```

1:50

Word 0

Word 1

Word 2

Word 3

Word 4

Clicked! Word 5

Word 6

Word 7

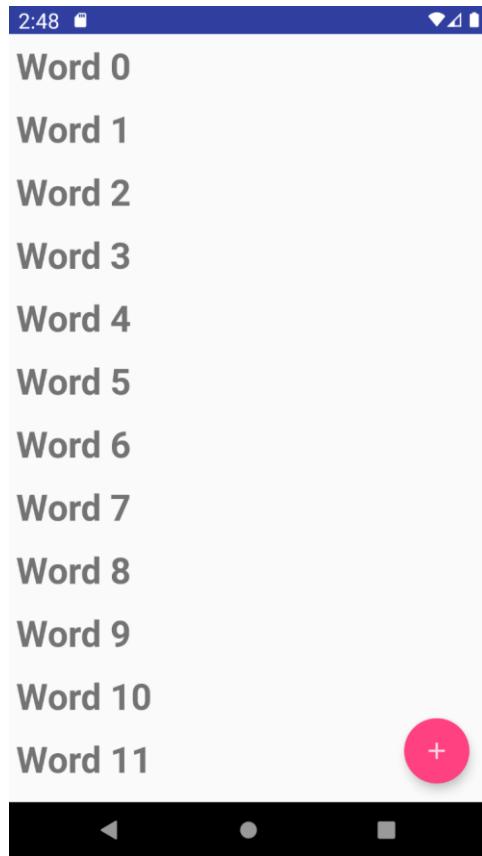
Word 8

Word 9

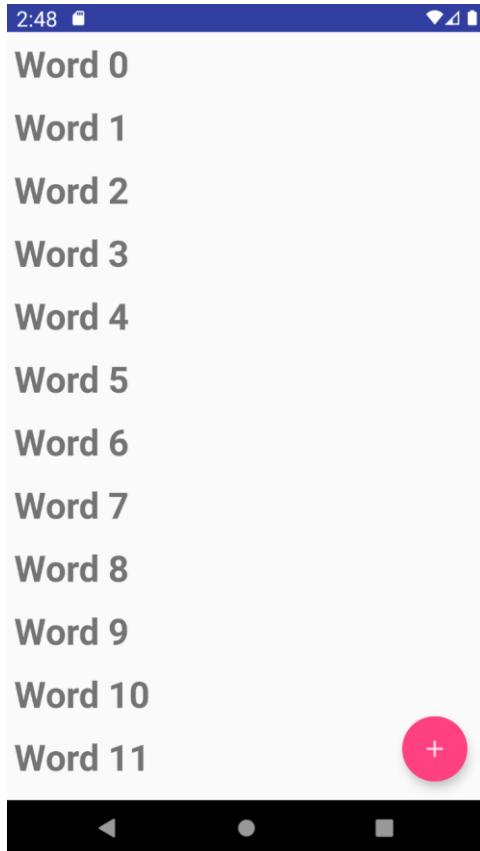
Word 10

Word 11

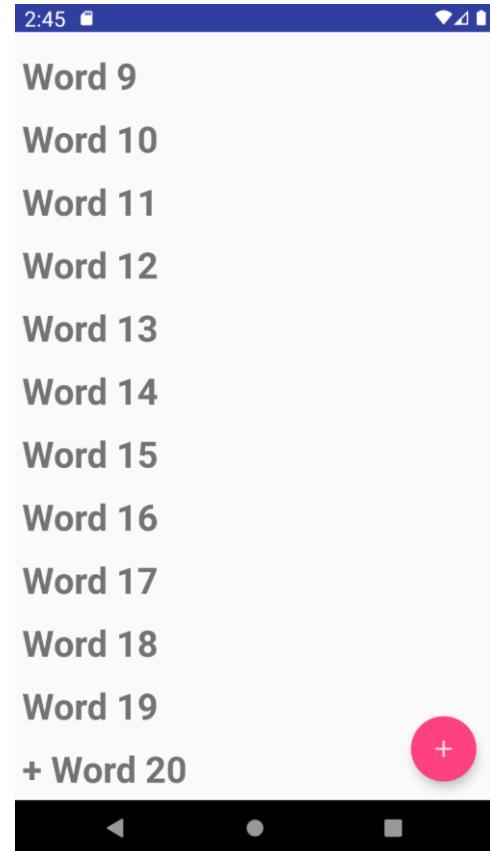
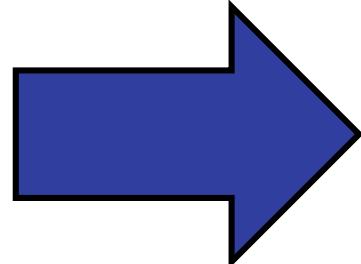
# Add More Interactivity!



We add a Floating  
Action Button **FAB**  
to the UI



User clicks on FAB



## activity\_main.xml

```
<include  
    layout="@layout/content_main"/>  
  
<FloatingActionButton  
    android:id="@+id/fab"/>
```

## MainActivity

```
onCreate(      ){
    setContentView(activity_main)
}

}
```

## activity\_main.xml

```
<include
layout="@layout/content_main"/>

<FloatingActionButton
        android:id="@+id/fab"/>
```

## MainActivity

```
onCreate(      ){
    setContentView(activity_main)
    FloatingActionButton fab = findViewById(fab)

}
}
```

## activity\_main.xml

```
<include
    layout="@layout/content_main"/>

<FloatingActionButton
    android:id="@+id/fab"/>
```

## MainActivity

```
onCreate(    ){
    setContentView(activity_main)
    FloatingActionButton fab = findViewById(fab)
}

}
```

## activity\_main.xml

```
<include
layout="@layout/content_main"/>

<FloatingActionButton
    android:id="@+id/fab"/>
```



FloatingActionButton



FloatingActionButton

Anonymous Class of  
Type  
View.OnClickListener()

## FloatingActionButton

Anonymous Class of  
Type  
View.OnClickListener()

```
onClick(View view) {  
  
}
```

## FloatingActionButton

Anonymous Class of  
Type  
`View.OnClickListener()`

```
onClick(View view) {  
    int wordListSize = mWordList.size()  
}  
}
```

## FloatingActionButton

Anonymous Class of  
Type  
`View.OnClickListener()`

```
onClick(View view) {  
    int wordListSize = mWordList.size()  
    mWordList.addLast("+" Word " + wordListSize)  
}
```

```
onClick(View view) {  
    int wordListSize = mWordList.size()  
    mWordList.addLast("+ Word " + wordListSize)  
}
```

20	+ Word 20
19	Word 19
18	Word 18
17	Word 17
16	Word 16
15	Word 15
14	Word 14
13	Word 13

## FloatingActionButton

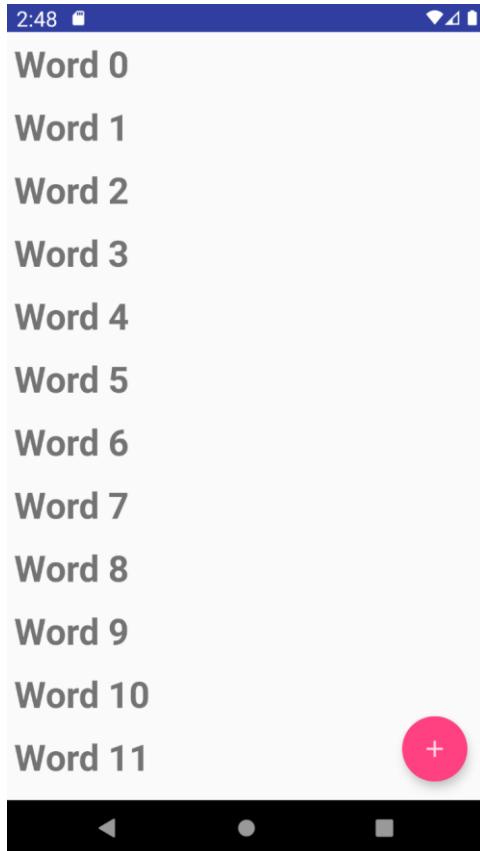
Anonymous Class of  
Type  
`View.OnClickListener()`

```
onClick(View view) {  
    int wordListSize = mWordList.size()  
    mWordList.addLast("+" Word " + wordListSize)  
    mRecyclerView.getAdapter().notifyItemInserted(wordListSize)  
}
```

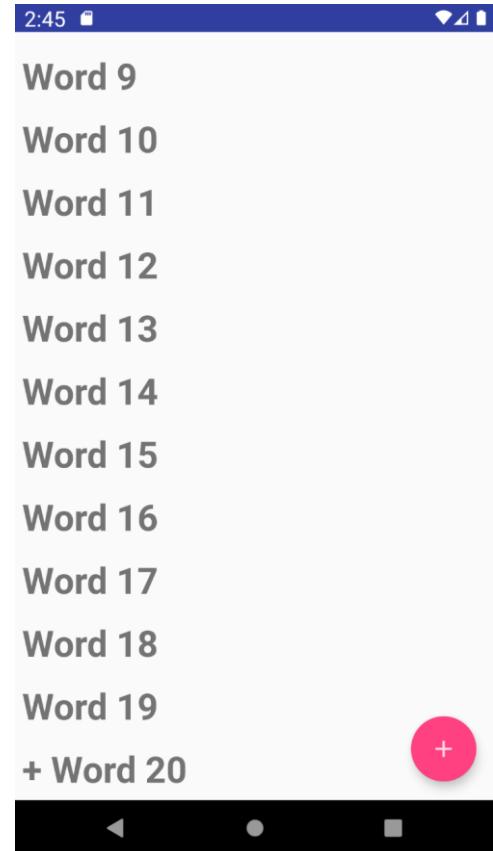
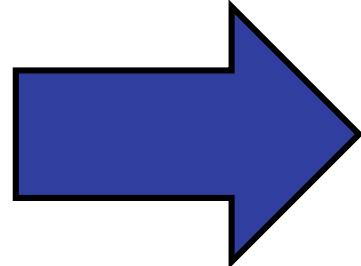
## FloatingActionButton

Anonymous Class of  
Type  
`View.OnClickListener()`

```
onClick(View view) {  
    int wordListSize = mWordList.size()  
    mWordList.addLast("+" Word " + wordListSize)  
    mRecyclerView.getAdapter().notifyItemInserted(wordListSize)  
    mRecyclerView.smoothScrollToPosition(wordListSize)  
}
```



User clicks on FAB



## MainActivity

```
mWordList
mRecyclerView

onCreate(      ){
    FloatingActionButton fab = findViewById(fab)
    fab.setOnClickListener(new View.OnClickListener() {

        public void onClick(View view) {
            int wordListSize = mWordList.size()
            mWordList.addLast("+" Word " + wordListSize)
            mRecyclerView.getAdapter().notifyItemInserted(wordListSize)
            mRecyclerView.smoothScrollToPosition(wordListSize)
        }
    })
}
```

# Learn More

- [RecyclerView](#)
- [RecyclerView class](#)
- [RecyclerView.Adapter class](#)
- [RecyclerView.ViewHolder class](#)
- [RecyclerView.LayoutManager class](#)

# Mobile Software Development

**CMPU 3036 DT211C/3, DT282/3**

# Today's Lecture

- Room Components
- Database Demo

# Annotations

- Annotations provide information about your program (classes, methods, fields) accessible to processing by **external tools**.
- Similar to **modifier** (e.g. public or static), should appear on own line before other modifiers.
- Indicated by an @ character followed by name of annotation type.

# Example

- Keeping track of **code reviews**
- Annotations allow you to provide review information e.g. reviewer name, date

```
@Reviewed(reviewer = "Jane Smith", date = 20181109)
public class MyClass {

    // ...
}
```

# Example

- Values are supplied for **elements** of annotation type e.g. reviewer = “Jane Smith”
- Tool can then automatically **extract** these values from either source file, or (more often) compiled class file.  
e.g. determine whether a class has been reviewed since it was last **modified**.



A ROOM

~~YOU  
DIDN'T SHOULD GET~~

# Overview

- Room is a robust SQL **object** mapping library
- Generates **SQLLite** Android code
- Provides a **simple** API for your database

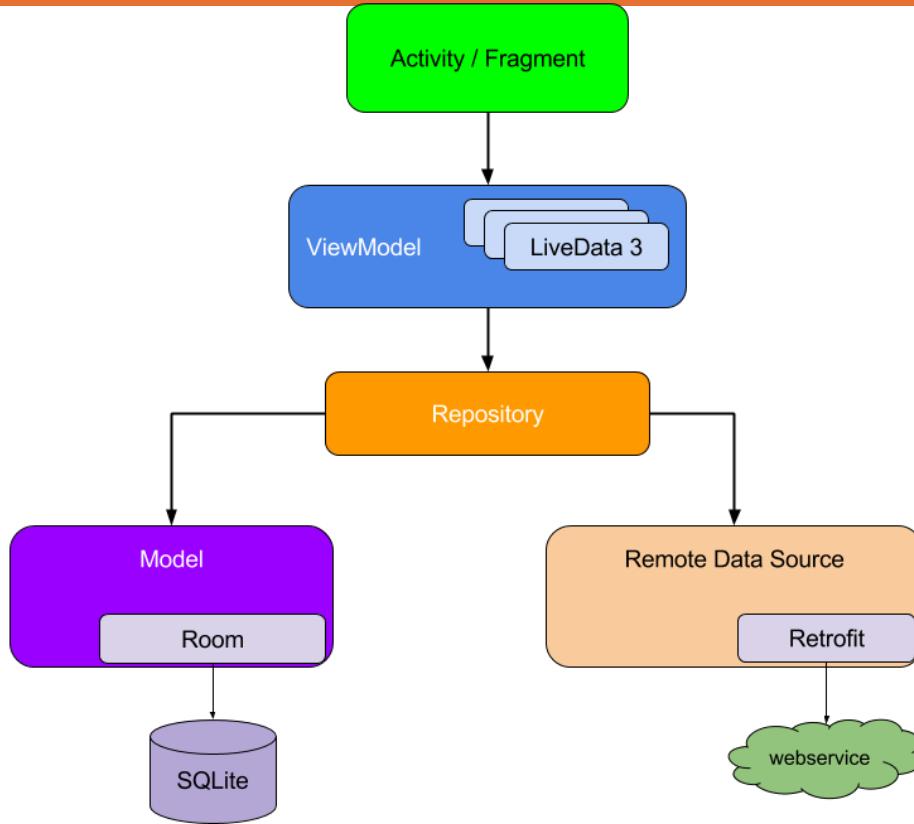
# Overview

- Offers layer over SQLite for smooth database access while harnessing full power of SQLite.
- Apps that handle reasonable amounts of structured data can benefit greatly from storing that data **locally**.
- Most common use case is **caching** relevant pieces of data i.e. when device cannot access network, user can still browse content **offline**.

# Overview

- User-initiated content changes **synced** to server after device is back online.
- As Room takes care of these concerns for you, it is highly **recommended** using Room instead of SQLite.

# Modern Android

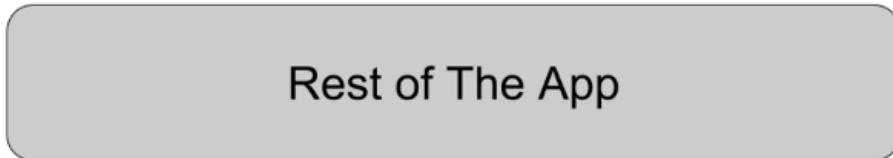


# Components

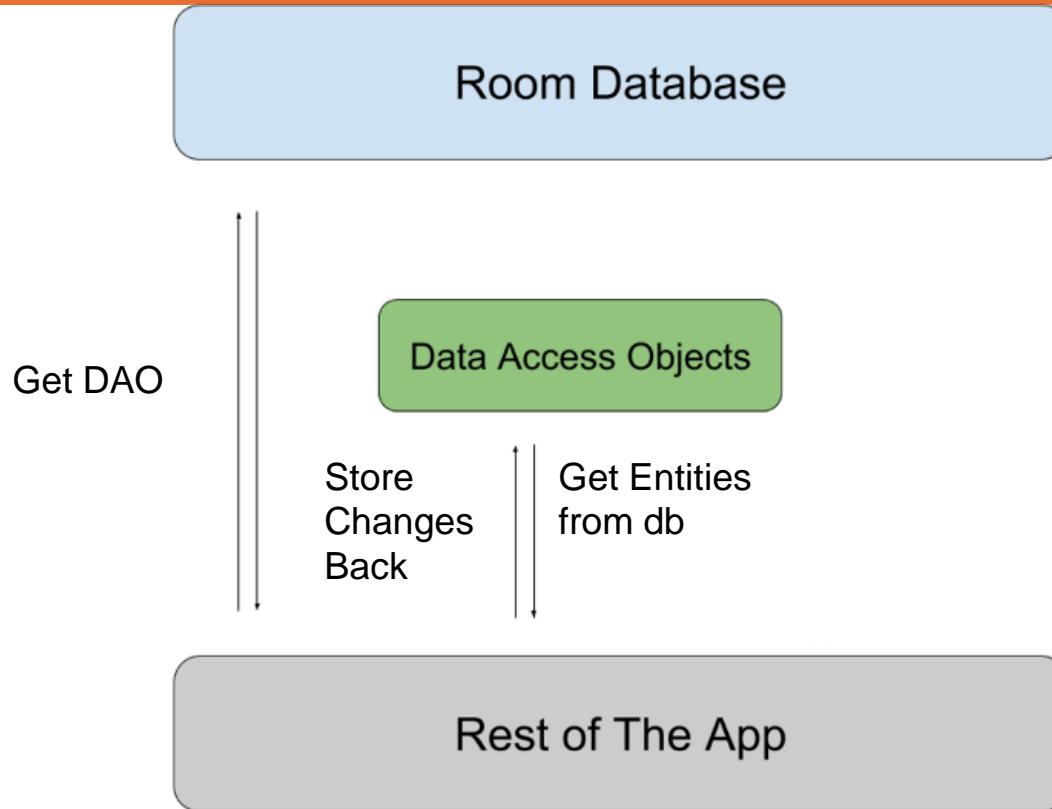
3 major components in Room:

- **Entity** Represents table within database.
- **Database** Contains database holder and serves as main **access point** for underlying connection to app's stored, relational data.
- **Data Access Objects (DAO)**: Contains methods used for accessing database.

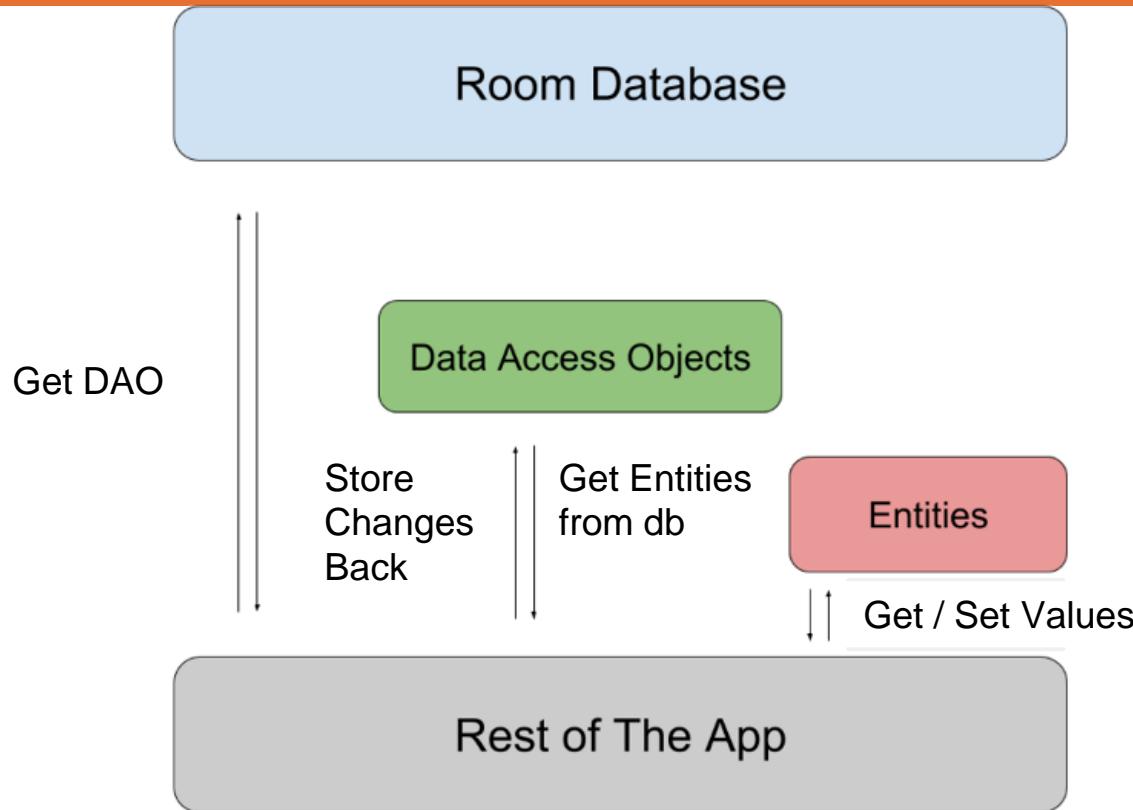
# Components



# Components



# Components



# Entities

- Sets of related fields are defined as entities.
- For each entity, **table** is created within database object to hold items.
- By default, Room creates a **column** for each field defined in entity.
- To store a field, Room must have **access** to it.
- 2 options make field **public**, or you can provide a getter and setter.

# Example

```
public class Person {  
    private int uid;  
    private String firstName;  
    private String lastName;  
}
```

<b>uid</b>	<b>firstName</b>	<b>lastName</b>
12345	Aleks	Becker
12346	Jhansi	Kumar

# Entities

Entities can have:

- An **empty** constructor
- Constructor whose parameters contain types and names that **match** those of fields in entity.
- **Partial** constructors, such as a constructor that receives only some of the fields.

# Primary Key

- Each entity must define at least 1 field as primary key.
- Even when only 1 field, must still be annotated with `@PrimaryKey`.
- To assign **automatic** IDs, set `@PrimaryKey`'s `autoGenerate` property.
- If entity has **composite** primary key, use the `primaryKeyes` property of the `@Entity` annotation.

# Names

- **By default**, Room uses class name as database table name.
- For different name, set tableName property of @Entity
- Caution: table names in SQLite are **case-insensitive**.
- Room uses field names as **column** names in database.
- Again, for a different name, add @ColumnInfo annotation to field.

# Example

```
@Entity  
public class Person {  
    @PrimaryKey (autoGenerate=true)  
    private int uid;  
  
    @ColumnInfo(name = "first_name")  
    private String firstName;  
  
    @ColumnInfo(name = "last_name")  
    private String lastName;  
  
    // + getters and setters if variables are private.  
}
```

# Names

## @NonNull

- Denotes parameter, field, or method return value can **never** be null
- Use for mandatory fields
- Primary key **must** use @NonNull



# Data Access Objects

- To **access** app's data we work with data access objects, or DAOs. #
- The set of Dao objects forms main component of Room, as each DAO includes methods that offer **abstract** access to your app's database.
- Why use DAO class instead of query builders or direct queries?
  1. **Separate** different components of your database architecture.
  2. Easily **mock** database access to test app.

# Data Access Objects

- A DAO can be either **interface** or abstract class.
- If abstract class, can optionally have constructor that takes a RoomDatabase as only parameter.
- Room creates each DAO implementation at **compile** time.
- **Note** Room doesn't support database access on Main Thread unless allowMainThreadQueries() called on builder.

# Insert

- When you create a DAO method and annotate it with `@Insert`
- Room generates an implementation that inserts all parameters into the database in a single transaction.

# Insert

```
@Dao
public interface MyDao {
    @Insert
    public void insertUsers(User... users);

    @Insert
    public void insertBothUsers(User user1, User user2);

    @Insert
    public void insertUsersAndFriends(User user, List<User> friends);
}
```

# Update

- Update method modifies a set of entities, given as parameters
- It uses a query that matches against primary key of each entity.

```
@Dao
public interface MyDao {
    @Update
    public void updateUsers(User... users);
}
```

# Delete

- Delete convenience method removes a set of entities, given as parameters, from database.
- It uses the primary keys to find the entities to delete.

```
@Dao
public interface MyDao {

    @Delete
    public void deleteUsers(User... users);
}
```

# Queries

- @Query is main annotation used in DAO classes.
- It allows you to perform read/write operations on database.
- Each @Query method is verified at compile time
  - i.e. if there is problem with query, **compilation error** occurs instead of **runtime failure**.

# Queries

- This is a query that loads all users.
- At compile time, Room knows it is querying all columns in user table.

```
@Dao  
public interface MyDao {  
  
    @Query("SELECT * FROM user")  
    public User[] loadAllUsers();  
}
```

# Queries

- If query contains syntax error or user table doesn't exist
- Room displays error with appropriate message as app compiles.

```
@Dao  
public interface MyDao {  
  
    @Query("SELECT * FROM user")  
    public User[] loadAllUsers();  
}
```

# Queries

- Passing parameters into queries performs filtering operations  
e.g. as displaying only users who are older than a certain age.

```
@Dao
public interface MyDao {

    @Query("SELECT * FROM user WHERE age > :minAge")
    public User[] loadAllUsersOlderThan(int minAge);

}
```

# Queries

- You can also pass multiple parameters in a query.

```
@Dao
public interface MyDao {

    @Query("SELECT * FROM user WHERE age BETWEEN :minAge AND :maxAge")
    public User[] loadAllUsersBetweenAges(int minAge, int maxAge);
}
```

# More Examples

```
@Query("DELETE FROM word_table")
void deleteAll();

@Query("SELECT * from word_table ORDER BY word ASC")
List<Word> getAllWords();

@Query("SELECT * FROM word_table WHERE word LIKE :word ")
public List<Word> findWord(String word);
```

# Database

The class annotated with `@Database` should:

1. Be **abstract** class that extends `RoomDatabase`.
2. Include list of **entities** associated with database within annotation.
3. Contain abstract method with 0 arguments and return class annotated with `@Dao`.

```
@Database(entities = {Word.class}, version = 1)
public abstract class WordRoomDatabase extends RoomDatabase {

    public abstract WordDao wordDao();
```

# Database

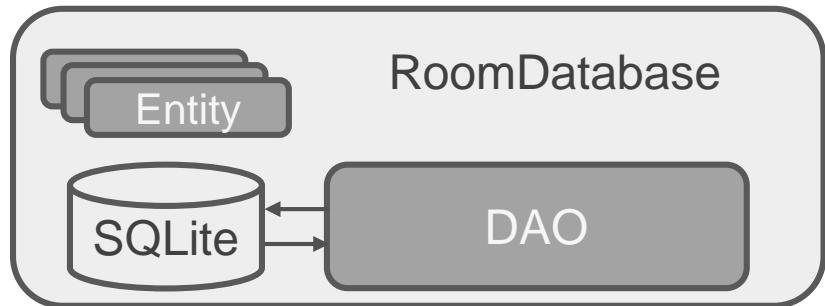
At **runtime**, an instance of Database is obtained by calling:

Room.databaseBuilder() with class and database name

```
INSTANCE = Room.databaseBuilder(context,  
    WordRoomDatabase.class, "word_database")  
    //...  
  
.build();
```

# Summary

- Room works with DAO and Entities
- Entities define the database schema
- DAO provides methods to access database



# Mobile Software Development

**CMPU 3036 DT211C/3, DT282/3**

# Today's Lecture

- Android's File System

# Overview

- Android uses file system similar to disk-based ones on other platforms.
- If familiar with **Unix-like** systems, you will understand file hierarchy very well.
- In Linux, file hierarchy is single tree, with **root** denoted as /
- Customised version of existing Linux hierarchy.

# Top Level Directories

## **/cache**

- Frequently accessed data and app components.

## **/data**

- Contains data of each application.
- Most data belonging to user e.g. contacts, SMS, dialed numbers, etc.

# Top Level Directories

## **/data**

- Private data of all apps.

## **/proc**

- Provides access to OS data structures.
- Several programs use /proc as source for information.
- Contains files that have useful information about processes.

# Top Level Directories

## /sdcard

- Data present on SD card of device.
- Any app with

WRITE\_EXTERNAL\_STORAGE

permission may create files or folders in this location.

## /system

- Libraries, system binaries, and other system-related files.

# Top Level Directories

## **system/app**

- System apps and preinstalled apps.
- Mounted as read only to prevent any changes.

# Types of Storage

- All Android devices have two file storage areas:
  1. Internal
  2. External
- Names from early days of Android, when most devices offered:
  1. Built-in **Non-Volatile** memory (internal storage)
  2. A **Removable** storage medium e.g. micro SD card (external storage)

# Internal Storage



# Internal Storage

- Internal files typically located in app's **/data/data** subdirectory.
- Data stored here is **private** and cannot be accessed by other apps.
- Even device owner is prevented from viewing files (unless they have **root access**).

# /data/data

1. **lib** - Custom library files required by app
2. **files** - Developer-saved files
3. **cache** - Files cached by app
4. **databases** - SQLite and journal files
5. **shared\_prefs** - XML file of shared preferences
6. Folders other than these are custom folders created by app developer.

# Internal Storage

1. Internal storage best option if neither user nor other apps should access files.
2. Always **available**
3. Only your app can access files, unless explicitly set to be readable or writable.
4. On app **uninstall**, system removes all app's files from internal storage

# Saving Files to Internal Storage

1. App's internal storage directory is specified by app's package name and can be accessed through Android's APIs.
2. Note: Unlike the external storage directories, your app does not require any system **permissions** to read and write to internal directories.

# Code

1. A **File** object works well for reading or writing large amounts of data in start-to-finish order without skipping around e.g. image files, network data.
2. The exact location where files can be saved might vary across devices, so use **API methods** to access storage instead of absolute file paths.

# Writing a File

When saving file to internal storage, you can access **directory** as File by calling either of the following methods:

1. `getFilesDir()` - returns a File representing internal **directory** for app.
2. `getDir(name, mode)` - Creates new **directory** (or opens an existing directory) within your app's unique file system directory.

# Writing a File

- To create a new **file** in one of these directories, use `File()` constructor, passing `File` provided by one of above methods that specifies your internal storage directory.

```
File file = new File(context.getFilesDir(), filename);
```

- A 3<sup>rd</sup> way is to call `openFileOutput()` to get a `FileOutputStream` that writes to a file in your internal directory.

# 1st Way

```
// Want to write this text to a file
String fileContents = "Hello world!\n";

// Get default internal directory
File d1 = getFilesDir();

// Create a file in this directory
File f1 = new File(d1, "myfile1");
```

# 1st Way

```
try {  
    FileOutputStream stream = new FileOutputStream(f1);  
  
    OutputStreamWriter writer = new OutputStreamWriter(stream);  
    BufferedWriter buf = new BufferedWriter(writer);  
  
    buf.write(fileContents);  
  
    buf.flush(); buf.close(); writer.close(); stream.close();  
} catch (Exception e) {  
}
```

# Results

```
/data/data/com.example.myfiles  
ls  
cache code_cache files  
cd files  
ls  
myfile1  
cat myfile1  
Hello world!
```

# 2<sup>nd</sup> Way

```
// Want to write this text to a file
String fileContents = "Hello world!\n";

// Create a directory called stuff
File d2 = getDir("stuff", MODE_PRIVATE);

// Create a file in that directory
File f2 = new File(d2, "myfile2");
```

# 2<sup>nd</sup> Way

```
try {  
    FileOutputStream stream = new FileOutputStream(f2);  
    OutputStreamWriter writer = new OutputStreamWriter(stream);  
    BufferedWriter buf = new BufferedWriter(writer);  
  
    buf.write(fileContents);  
  
    buf.flush(); buf.close(); writer.close(); stream.close();  
} catch (Exception e) {  
}
```

# Results

```
cd ..  
cd app_stuff/  
ls  
myfile2  
cat myfile2  
Hello world!
```

# 3<sup>rd</sup> Way

```
// Want to write this text to a file  
String fileContents = "Hello world!\n";  
  
// Give the file a name  
String f3 = "myfile3";
```

# 3<sup>rd</sup> Way

```
try {  
    FileOutputStream outputStream = openFileOutput(f3, Context.MODE_PRIVATE);  
  
    outputStream.write(fileContents.getBytes());  
  
    outputStream.flush();  
    outputStream.close();  
} catch (Exception e) {  
}  
}
```

# Results

```
cd ..  
cd files  
ls  
myfile1 myfile3  
cat myfile3  
Hello world!
```

# Writing a File

- `openFileOutput()` takes mode parameter e.g. `MODE_PRIVATE` makes it private to your app.
- Other apps **cannot** browse your internal directories and do not have read or write access unless you explicitly set files to be readable or writable.

# Cache

- `getCacheDir()` - returns File representing internal directory for your app's temporary cache files.
- To create private cache files use `createTempFile()`
- Caution: If system runs low on storage, it may delete cache files without warning (i.e. always check for their existence)
- Delete each file once it is no longer needed.

# Cache

```
try {  
    // Get the cache directory  
    File theCache = getCacheDir();  
  
    // Create a temporary file in this directory  
    File cached = File.createTempFile("myfile4", "", theCache);  
  
}catch (Exception e)  
{  
}
```

# Results

```
cd ..  
cd cache  
ls  
myfile4433013072222818995
```

# Reading a File

- To read an existing file in the **default** storage location, call `openFileInput(name)`, passing name of file.

# Reading a File

```
try { FileInputStream fs = openFileInput("myfile3");

    StringBuilder temp = new StringBuilder();
    if (fs != null) {
        InputStreamReader reader = new InputStreamReader(fs);
        BufferedReader buf = new BufferedReader(reader);
        String line;
        while ((line = buf.readLine()) != null) {
            temp.append(line + "\n");
        }
    }
} catch (Exception e){}
```

# Reading a File

- You can get String array of an app's file names in the **default** storage location by calling `fileList()`.

```
String[] myfiles = fileList();

for(int i = 0; i < myfiles.length; i++){
    // do something with myfiles[i]
}
```

# Reading a Raw File

- To package file in app that is accessible at run time, save it in project's res/raw/ directory.
- You can open these files with `openRawResource()`, passing the `R.raw.filename` resource ID.
- Method returns `InputStream` to read the file.
- You cannot write to original file.

# External Storage



# External Storage

- Place for files that:
  1. Don't require access restrictions,
  2. Can be shared with other apps,
  3. Allow user to access externally from computer.
- Not always available e.g. user can mount external storage as USB
- World-readable - files may be read outside of your control.
- If app uninstalled, system removes files **only** if saved in directory from `getExternalFilesDir()`

# Saving a File

- After you verify that storage is **available**, you can save two different types of files:

# External Public

Public files:

- Files that should be freely available to other apps and user.
- If user uninstalls app, these files **remain** available to user e.g. photos captured by app or other downloaded files should be saved as public files.

# External Private

Private files:

- Files that rightfully belong to your app and will be **deleted** if user uninstalls app.
- Although files technically accessible by user and other apps because on external storage, they don't provide value outside of your app.

# External Storage

- Caution: External storage might become unavailable if user removes SD card.
- Files are visible to user and other apps that have `READ_EXTERNAL_STORAGE` permission.
- If app's functionality depends on files or you need to completely restrict access, write them to internal storage instead.

# Permissions

- To write to public external storage, request the WRITE\_EXTERNAL\_STORAGE permission in manifest file: (this also grants permission to read external storage)

```
<manifest ...>  
    <uses-permission  
        android:name="android.permission.WRITE_EXTERNAL_STORAGE" />  
</manifest>
```

# Permissions

- If app only needs to read external storage (but not write to it), then request READ\_EXTERNAL\_STORAGE permission:

```
<manifest ...>  
    <uses-permission  
        android:name="android.permission.READ_EXTERNAL_STORAGE" />  
</manifest>
```

# Permissions

- Reading or writing files in app's **private** external storage directory - accessed via `getExternalFilesDir()`—does **not** require permissions.

# Availability

- You can query external storage state with `getExternalStorageState()`
- If returned state is `MEDIA_MOUNTED`, then you can read and write files.
- If it's `MEDIA_MOUNTED_READ_ONLY`, you can only read files.

# Availability

```
// Checks if external storage is available for read and write

public boolean isExternalStorageWritable() {

    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}
```

# Availability

```
/* Checks if external storage is available to at least read */

public boolean isExternalStorageReadable() {

    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state) ||
        Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        return true;
    }
    return false;
}
```

# Directory Categories

- Important to use directory names provided by API constants such as DIRECTORY\_PICTURES.
- This ensures files are treated properly by system e.g. files saved in DIRECTORY\_RINGTONES are categorised by system media scanner as ringtones not music.

# Public Directory

- To save public files on external storage, use `getExternalStoragePublicDirectory()` to get a `File` for appropriate directory.
- It takes argument specifying type of file so that they can be organised with other public files e.g. `DIRECTORY_MUSIC` or `DIRECTORY_PICTURES`.

# Public Directory

```
public File getPublicAlbumStorageDir(String albumName) {  
  
    // Get the directory for the user's public pictures directory.  
    File file = new File(Environment.getExternalStoragePublicDirectory(  
        Environment.DIRECTORY_PICTURES), albumName);  
  
    if (!file.mkdirs()) {  
        Log.e(LOG_TAG, "Directory not created");  
    }  
    return file;  
}
```

# Private Directory

- To save files that are **private** to your app call `getExternalFilesDir()` and pass it name of **type** of directory.
- A directory created this way is added to parent directory that contains all app's external storage files, which system deletes if app uninstalled.

# Private Directory

```
public File getPrivateAlbumStorageDir(Context context, String albumName)

    // Get the directory for the app's private pictures directory.
    File file = new File(context.getExternalFilesDir(
        Environment.DIRECTORY_PICTURES), albumName);

    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

# Private Directory

- If none of pre-defined directory names suit, call `getExternalFilesDir()` and pass **null**.
- This returns **root** directory for your app's private directory on external storage.

# Multiple Storage Locations

- Many devices now divide permanent storage space into separate "internal" and "external" partitions.
- So even without removable storage, these two storage spaces always exist.
- As external storage might be removable, there are some differences between these two options.

# Multiple Storage Locations

- Both locations accessed by calling `getExternalFilesDirs()`, which returns `File` array with entries for each storage location.
- First entry is considered primary external storage, and should be used unless full or unavailable.

# Query Free Space

- If size of data known, query whether enough space available without causing IOException by calling these methods on File object.
  - getFreeSpace()
  - getTotalSpace()
- Alternatively, try writing file and catch IOException if it occurs.

# Delete a File

- Always delete files that app no longer needs.
- Easiest way is to call `delete()` on File object e.g. `myFile.delete()`
- If file saved on internal storage, you can also ask the Context to locate and delete it by calling `myContext.deleteFile(fileName)`

# Delete a File

- If app uninstalled, system deletes:
- ALL files saved on internal storage.
- ALL files saved on external storage using `getExternalFilesDir()`
- Best practice to manually delete all cached files created with `getCacheDir()` on a regular basis and other files app no longer needs.

# Mobile Software Development

**CMPU 3036 DT211C/3, DT282/3**

# Today's Lecture



Location,  
Location,  
Location!

# Geographical Co-ordinates

- To **locate** object on Earth's surface, geographical coordinates needed.

# Geographical Co-ordinates

- To **locate** object on Earth's surface, geographical coordinates needed.
- Most common solution assigns 3 **coordinates** to each location:  
latitude, longitude and altitude (km above sea level).

# Geographical Co-ordinates

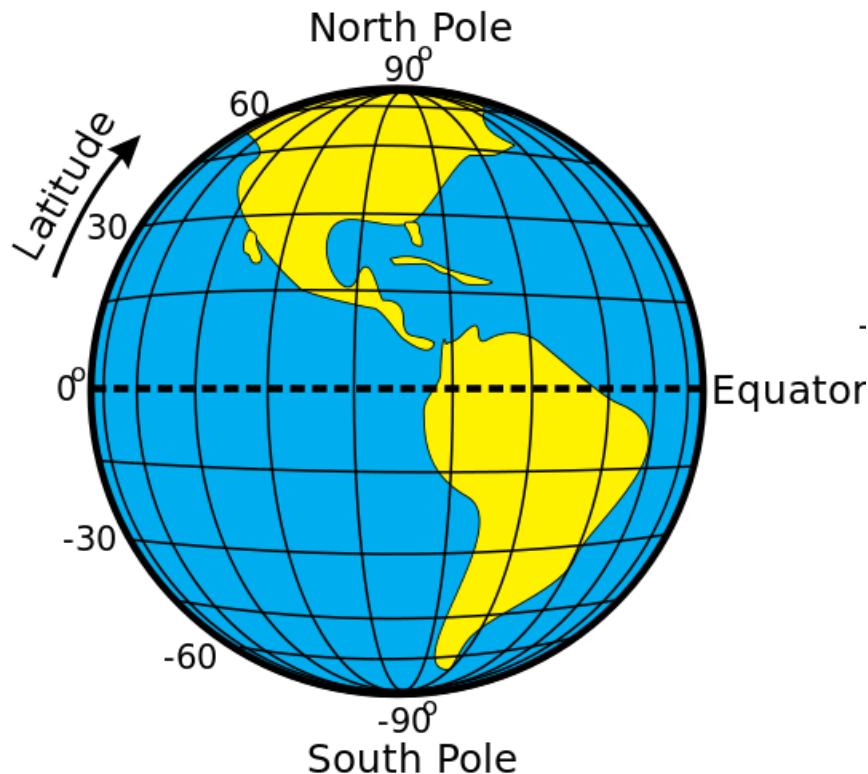
- To **locate** object on Earth's surface, geographical coordinates needed.
- Most common solution assigns 3 **coordinates** to each location:  
latitude, longitude and altitude (km above sea level).
- Earth **assumed** to be perfect sphere with known radius R.

# Geographical Co-ordinates

- To **locate** object on Earth's surface, geographical coordinates needed.
- Most common solution assigns 3 **coordinates** to each location:  
latitude, longitude and altitude (km above sea level).
- Earth **assumed** to be perfect sphere with known radius R.
- Latitude and longitude represents **angles** (in degrees) of location on  
surface, with respect to Earth centre.

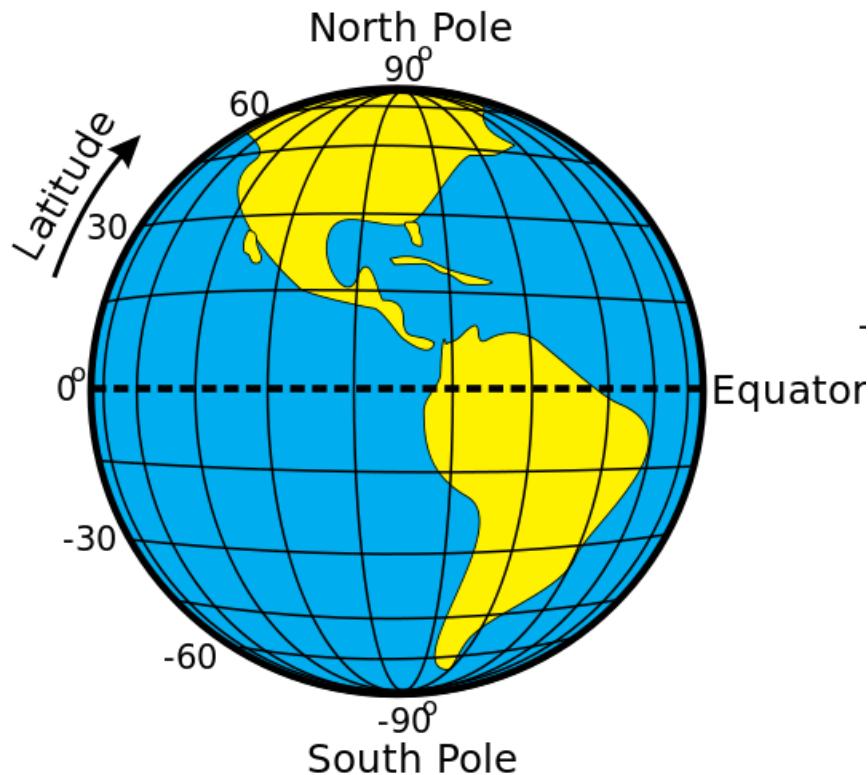
# Latitude

- Equator:  $0^\circ$
- North Pole:  $90^\circ$
- South Pole:  $-90^\circ$



# Latitude

- Equator:  $0^\circ$
- North Pole:  $90^\circ$
- South Pole:  $-90^\circ$
- Loosely speaking,  
latitude contributes to y  
(or **vertical**) component  
of location.



# Longitude

- Longitude measures x (or **horizontal**) component.

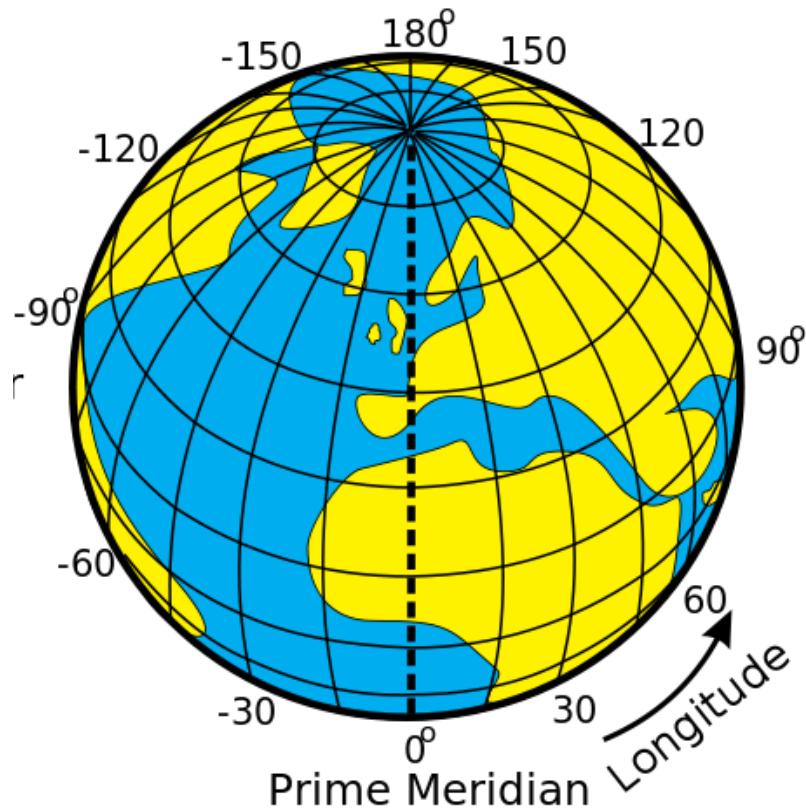
# Longitude

- Longitude measures x (or **horizontal**) component.
- Imaginary line that connects **Poles** passing through Greenwich called **Prime Meridian**.

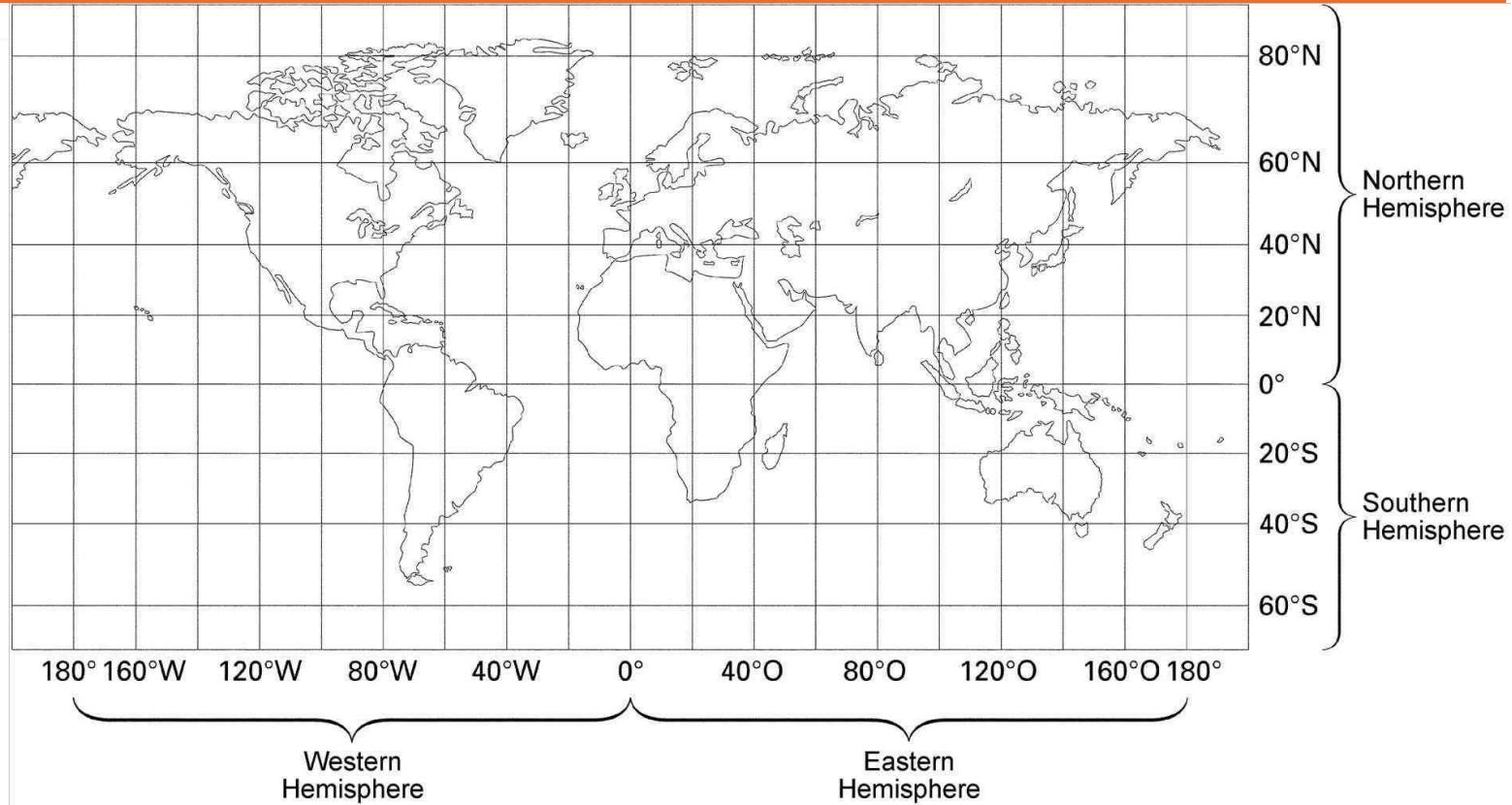


# Longitude

- Prime Meridian  $0^\circ$
- Increases **Eastwards** to Antipodal Prime Meridian at  $180^\circ$
- Decreases **Westwards** Antipodal Prime Meridian: at  $-180^\circ$



# Geographical Co-ordinates



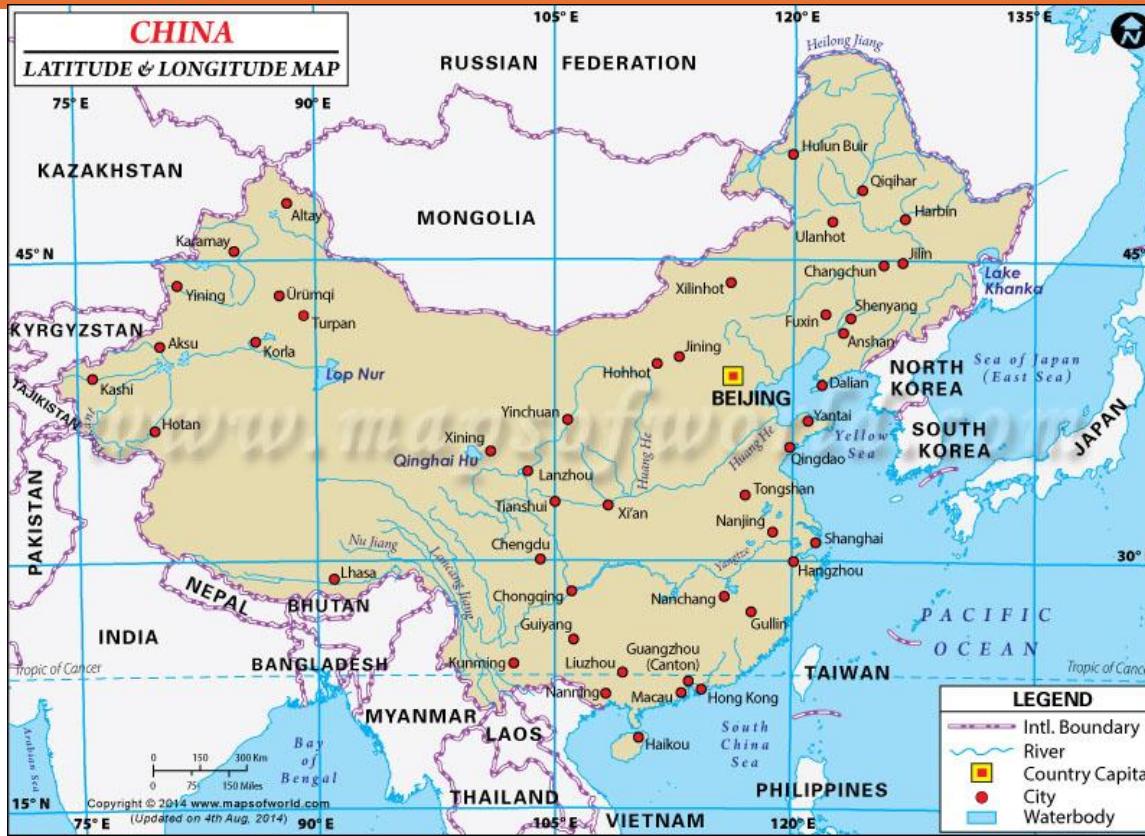
# Examples



# Examples



# Examples



# Examples



# Location

- Finding location in space of object **difficult**.

# Location

- Finding location in space of object **difficult**.
- No traditional sensor able to locate an object.

# Location

- Finding location in space of object **difficult**.
- No traditional sensor able to locate an object.
- What most sensors do is **estimate** distance from target object.

# Location

- Finding location in space of object **difficult**.
- No traditional sensor able to locate an object.
- What most sensors do is **estimate** distance from target object.
- Sonars and radars send out sound waves and calculate distance by detecting **how long** it takes for their signal to bounce back.
- Satellites basically do the same, but using radio waves.

# Solution

- Solution to locate object relies on **several** distance measurements, taken by different **sensors**.

# Solution

- Solution to locate object relies on **several** distance measurements, taken by different **sensors**.
- **Trilateration**, common technique used to calculate position of object given distance measurements.

# Solution

- Solution to locate object relies on **several** distance measurements, taken by different **sensors**.
- **Trilateration**, common technique used to calculate position of object given distance measurements.
- Note: Another technique called **Triangulation** works with angles instead of distances.

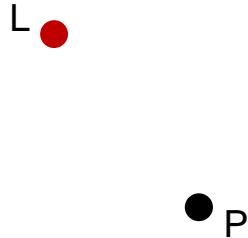
# How it Works

- Suppose we want to find location **P**



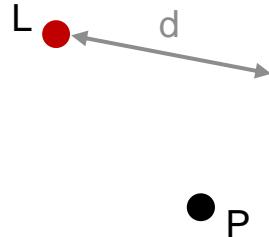
# How it Works

- Suppose we want to find location **P**
- Our 1<sup>st</sup> try uses beacon L, at **known** position.



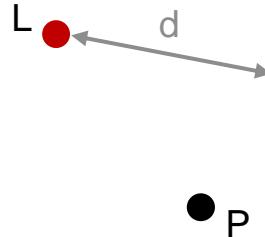
# How it Works

- Suppose we want to find location **P**
- Our 1<sup>st</sup> try uses beacon L, at **known** position.
- L can estimate its distance to P, d



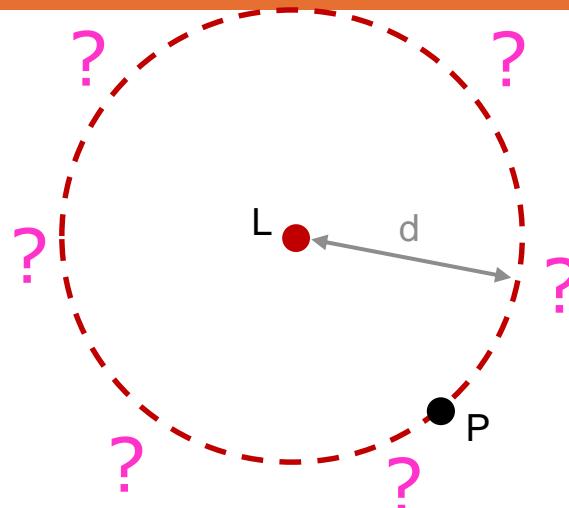
# How it Works

- Suppose we want to find location **P**
- Our 1<sup>st</sup> try uses beacon L, at **known** position.
- L can estimate its distance to P, d
- Cannot identify **exact** position of P.



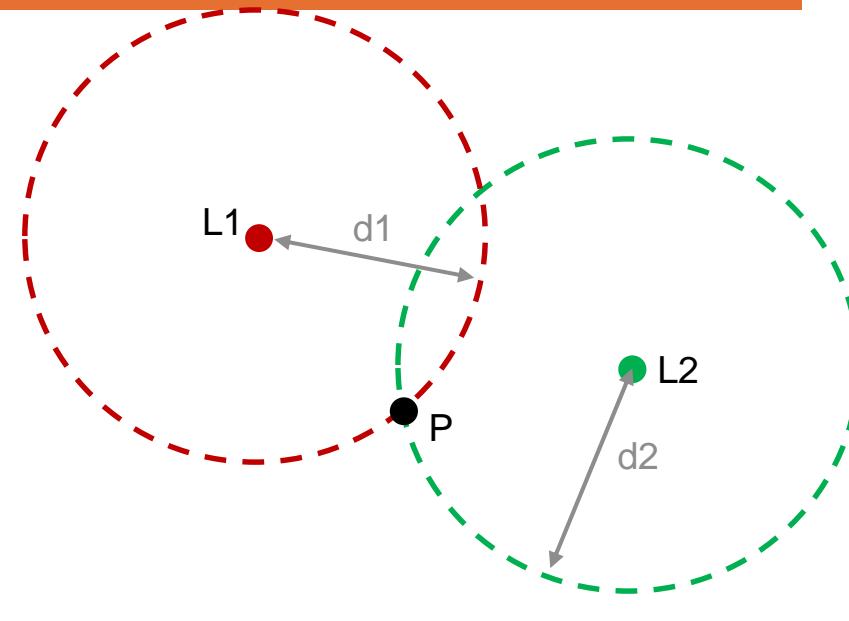
# How it Works

- Each point at distance  $d$  from  $L$  is **potential** candidate for  $P$ .
- Guess for  $P$  limited to circle of **radius  $d$**  around  $P$ .



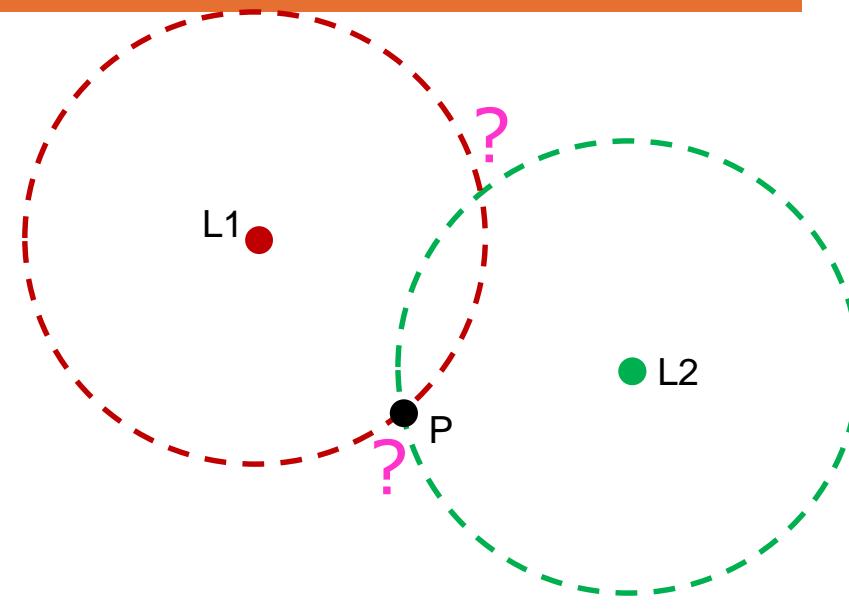
# How it Works

- Situation **improved** by using 2 beacons, L1 & L2.
- We **know** P lies along circumference of red circle.
- Similarly, **must** be on circumference of green circle.



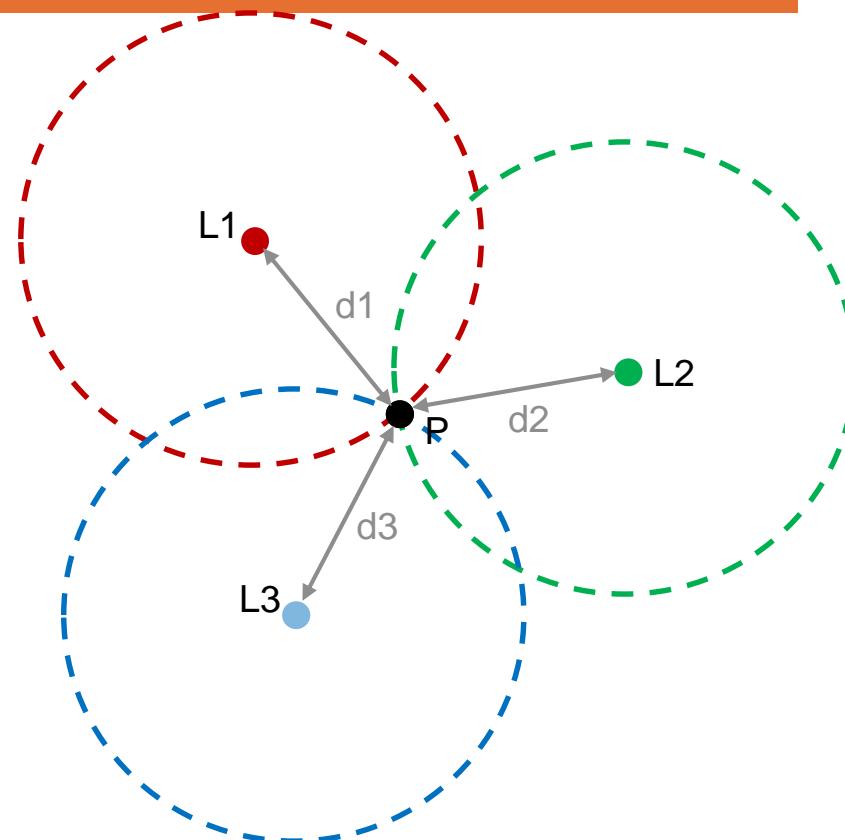
# How it Works

- Means it has to be at **intersections** of 2 circles.
- This suddenly restricts our guess to **only two** possible **locations**.



# How it Works

- To be absolutely precise, need a 3<sup>rd</sup> station, L3.
- 3 circles will meet at **only one point** i.e. location of P



# Global Positioning System GPS

- Constellation of 27 Earth-orbiting satellites 24 operational + 3 extras

# Global Positioning System GPS

- Constellation of 27 Earth-orbiting satellites 24 operational + 3 extras
- 1½ ton **solar-powered** units circle globe twice daily at 19,300 km.

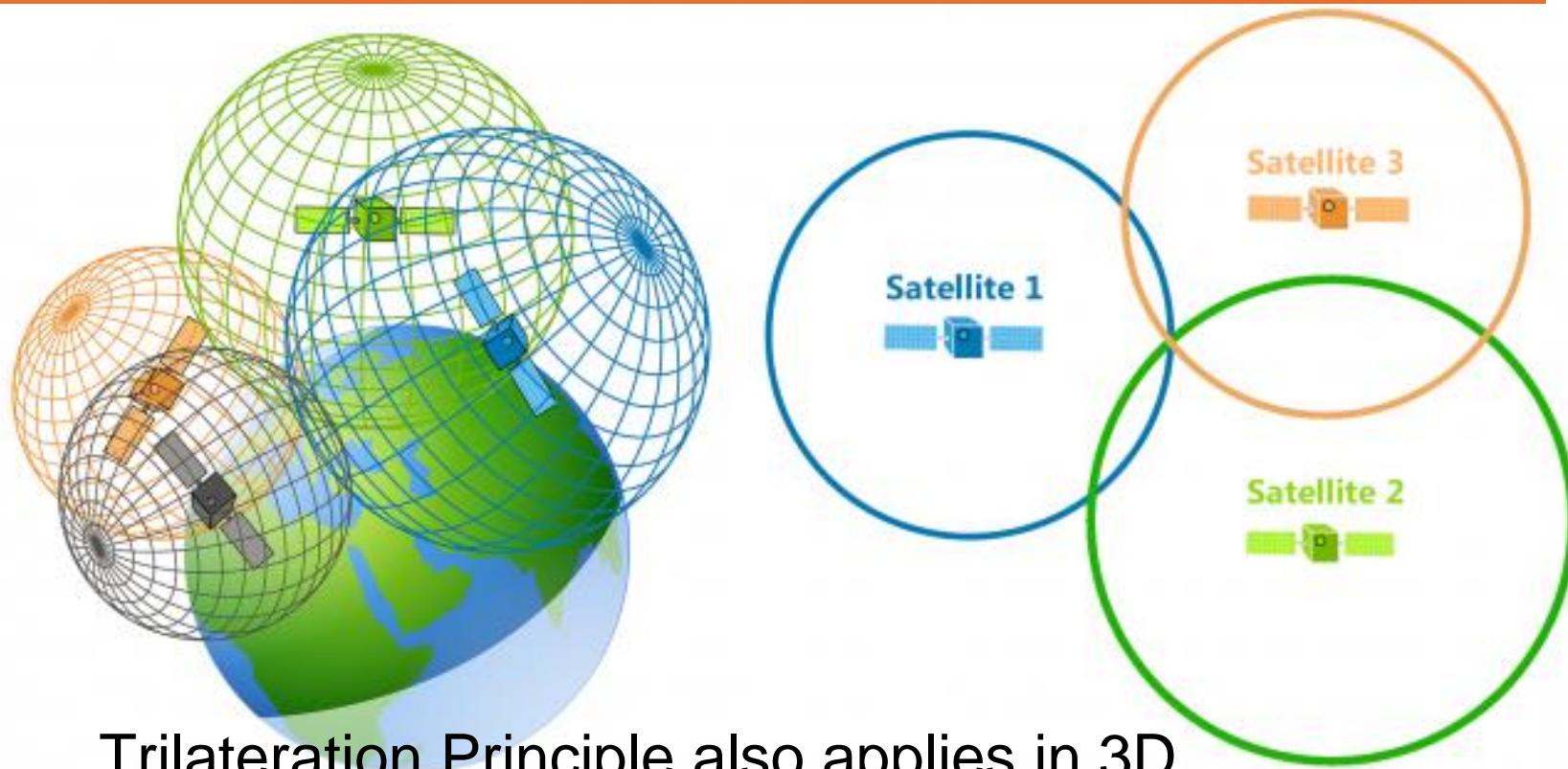
# Global Positioning System GPS

- Constellation of 27 Earth-orbiting satellites 24 operational + 3 extras
- 1½ ton **solar-powered** units circle globe twice daily at 19,300 km.
- Orbits arranged so, at any **time**, in any **place**, at least 4 satellites "visible" in sky.

# Global Positioning System GPS

- Constellation of 27 Earth-orbiting satellites 24 operational + 3 extras
- 1½ ton **solar-powered** units circle globe twice daily at 19,300 km.
- Orbits arranged so, at any **time**, in any **place**, at least 4 satellites "visible" in sky.
- GPS receiver's job to locate 4 or more of satellites, figure out **distance** to each, and use information to estimate its location.

# Global Positioning System GPS



Trilateration Principle also applies in 3D

# Nerdy Stuff

- At given time (e.g. midnight), satellite begins **transmitting** long, digital pattern called a pseudo-random code.

# Nerdy Stuff

- At given time (e.g. midnight), satellite begins **transmitting** long, digital pattern called a pseudo-random code.
- Receiver begins running **same** digital pattern also exactly at midnight.

# Nerdy Stuff

- At given time (e.g. midnight), satellite begins **transmitting** long, digital pattern called a pseudo-random code.
- Receiver begins running **same** digital pattern also exactly at midnight.
- When satellite's signal reaches receiver, its transmission of the pattern will **lag** a bit behind receiver's playing of pattern.

# Nerdy Stuff

- At given time (e.g. midnight), satellite begins **transmitting** long, digital pattern called a pseudo-random code.
- Receiver begins running **same** digital pattern also exactly at midnight.
- When satellite's signal reaches receiver, its transmission of the pattern will **lag** a bit behind receiver's playing of pattern.
- Length of **delay** is equal to signal's travel time.

# Nerdy Stuff Cont'd...

- Receiver multiplies this time by **speed of light** to determine how far signal travelled.

# Nerdy Stuff Cont'd...

- Receiver multiplies this time by **speed of light** to determine how far signal travelled.
- Assuming signal travelled in **straight line**, this is distance from receiver to satellite.

# Nerdy Stuff Cont'd...

- Receiver multiplies this time by **speed of light** to determine how far signal travelled.
- Assuming signal travelled in **straight line**, this is distance from receiver to satellite.
- NOTE: A GPS receiver has to have clear **line of sight** to satellite to operate, so dense tree cover and buildings can keep it from getting fix on location.

# WiFi Positioning System (WPS)

- WPS is geolocation system that uses characteristics of WiFi hotspots and other wireless access points to discover where device located.

# WiFi Positioning System (WPS)

- WPS is geolocation system that uses characteristics of WiFi hotspots and other wireless access points to discover where device located.
- Used when satellite navigation such as GPS inadequate:
  1. Multipath **interference** / signal blockage indoors.
  2. Establishing a satellite fix would take **too long**.

# WiFi Positioning System (WPS)

- WPS is geolocation system that uses characteristics of WiFi hotspots and other wireless access points to discover where device located.
- Used when satellite navigation such as GPS inadequate:
  1. Multipath **interference** / signal blockage indoors.
  2. Establishing a satellite fix would take **too long**.
- Viable due to **rapid growth** of wireless access points in urban areas.

# WiFi Positioning System (WPS)

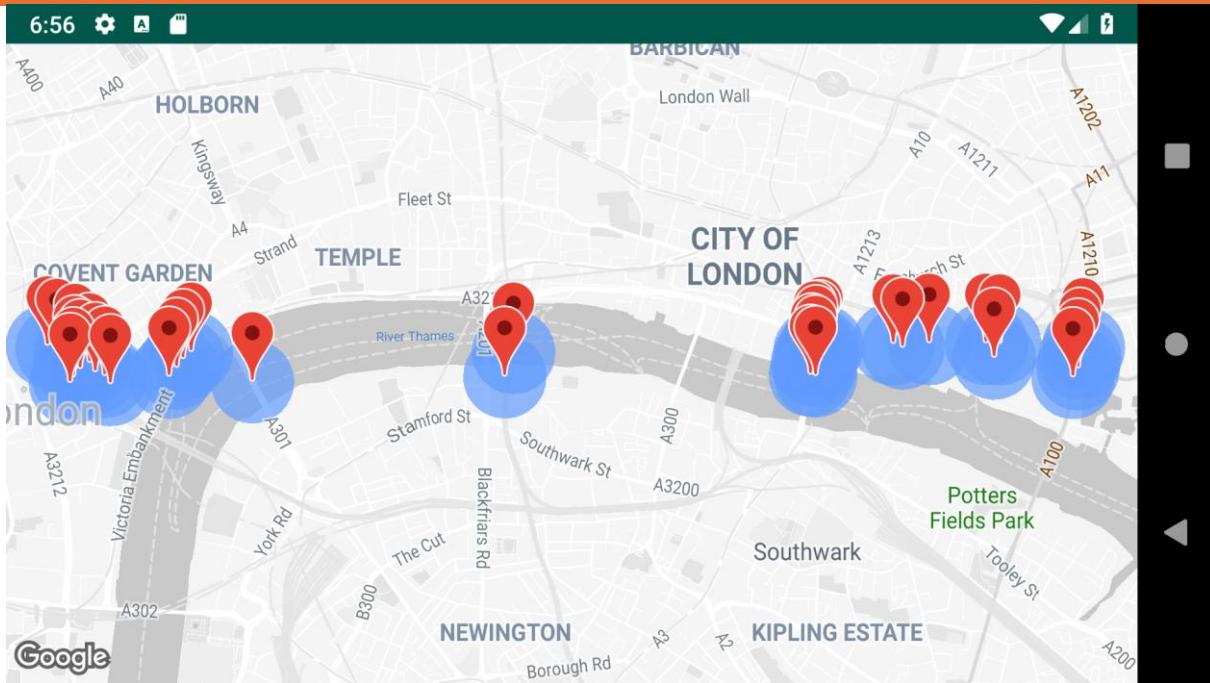
- Typical data useful to **geolocate** wireless access point include its SSID and MAC address.

bssid	lat	lon	created	updated
0014D1C3DE3E	-38.778	143.6603	1428513555	1428513555
0024B20B0C7E	-38.4141	143.6007	1428513555	1428513555
001B116CC29E	-38.3881	142.4835	1428513555	1428513555
001D7E4732AE	-38.3867	142.5014	1428513555	1428513555
081FF3239E25	-38.3865	142.4797	1428513555	1428513555

- Accuracy depends on number of nearby access points whose positions have been entered into **database**.

# Paranoid Android

Sample  
WiFi  
Hotspots  
London



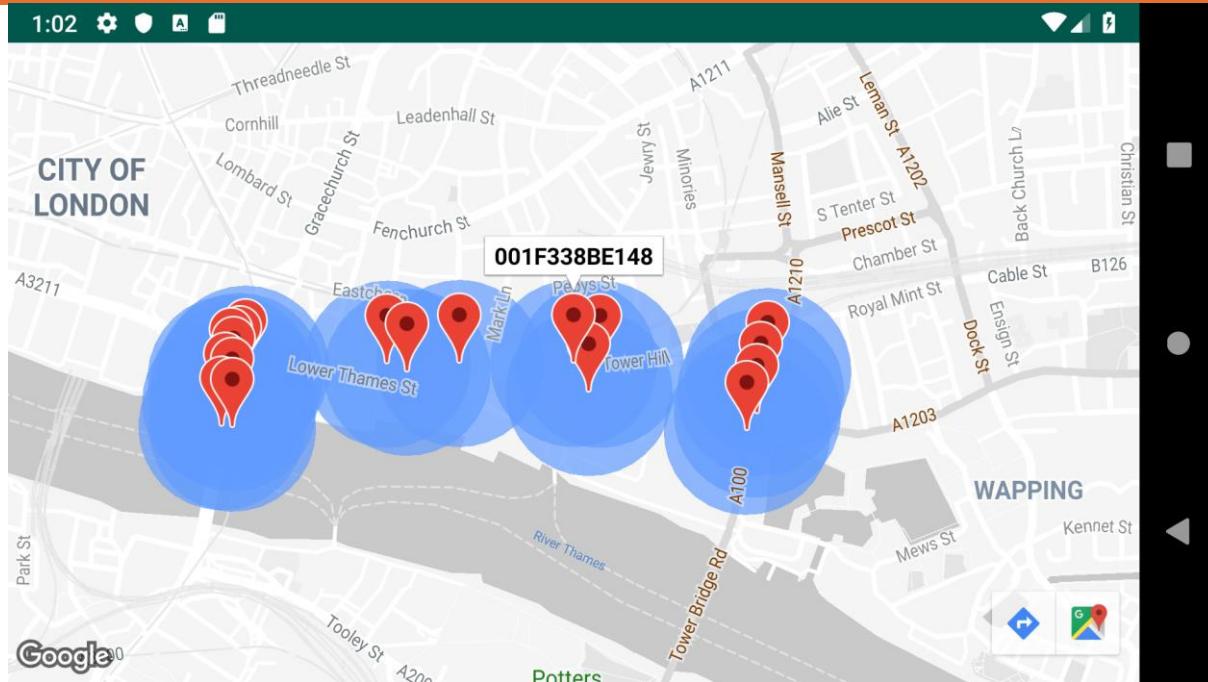
# WiFi Positioning System (WPS)

- Many techniques exist to determine **distance** or **angle** of client devices with respect to access points.
  1. Received signal strength indication (RSSI)
  2. Fingerprinting
  3. Angle of arrival (AoA)
  4. Time of flight (ToF)
- Trilateration or Triangulation **algorithms** then be used to determine position of target device.

# Paranoid Android

Database links BSSID to Latitude and Longitude location.

Then used with e.g. trilateration to get device position.



# Back to Android



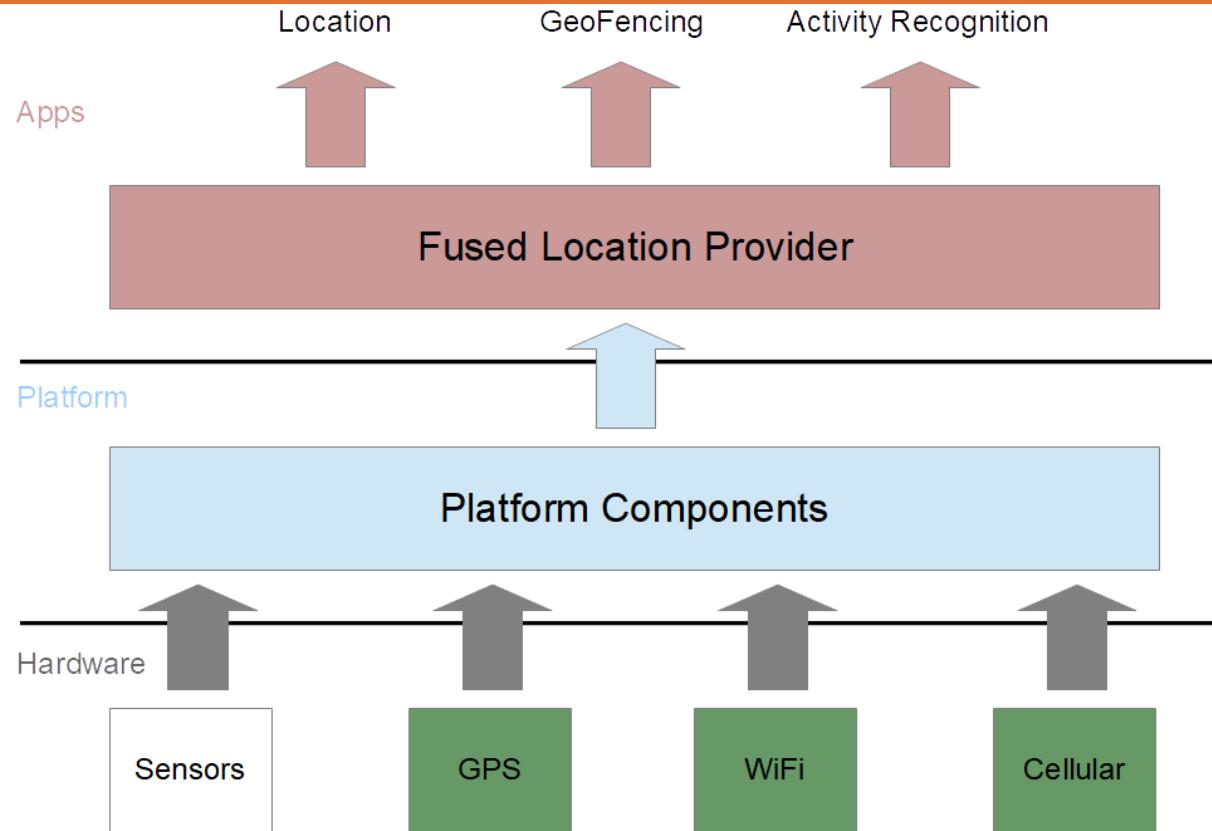
# Overview

- A unique feature of mobile apps is location **awareness**.
- Users take devices with them everywhere, so app can detect and use device location to customise experience for them.

# Overview

- A unique feature of mobile apps is location **awareness**.
- Users take devices with them everywhere, so app can detect and use device location to customise experience for them.
- Location **APIs** available in Google **Play** Services for adding:
  1. Location Tracking
  2. Geofencing
  3. Activity Recognition

# Overview



# Battery Drain...



Location gathering and battery drain are **directly** related as follows:

# Battery Drain...



Location gathering and battery drain are **directly** related as follows:

1. **Accuracy** - Precision of location data.

Higher accuracy, higher battery drain.

# Battery Drain...



Location gathering and battery drain are **directly** related as follows:

1. **Accuracy** - Precision of location data.

Higher accuracy, higher battery drain.

2. **Frequency** - How often location is computed.

Frequent location computations, more battery used.

# Battery Drain...



Location gathering and battery drain are **directly** related as follows:

1. **Accuracy** - Precision of location data.  
Higher accuracy, higher battery drain.
2. **Frequency** - How often location is computed.  
Frequent location computations, more battery used.
3. **Delay** - How quickly location data is delivered.  
Less delay requires more battery.

# Accuracy

In **Request** to FusedLocation API for location updates, use `setPriority()`

PRIORITY\_HIGH\_ACCURACY

Most accurate location. Computed using as many inputs as necessary e.g. GPS, Wi-Fi, cell and other sensors. May cause significant battery drain.

# Accuracy

In **Request** to FusedLocation API for location updates, use `setPriority()`

PRIORITY\_HIGH\_ACCURACY

Most accurate location. Computed using as many inputs as necessary e.g. GPS, Wi-Fi, cell and other sensors. May cause significant battery drain.

PRIORITY\_BALANCED\_POWER\_ACCURACY

Accurate location (within 100m) while optimising for power. Very rarely uses GPS. Typically combination of Wi-Fi and Cell information.

# Accuracy

In **Request** to FusedLocation API for location updates, use `setPriority()`

PRIORITY\_LOW\_POWER

Largely relies on cell towers and avoids GPS and Wi-Fi inputs, providing coarse (city-level, within 10km) accuracy with minimal battery drain.

# Accuracy

In **Request** to FusedLocation API for location updates, use `setPriority()`

PRIORITY\_LOW\_POWER

Largely relies on cell towers and avoids GPS and Wi-Fi inputs, providing coarse (city-level, within 10km) accuracy with minimal battery drain.

PRIORITY\_NO\_POWER

Receives locations passively from other apps for which location has already been computed.

# Accuracy

- Location needs of most apps can be **satisfied** using balanced power or low power options.
- High accuracy should be reserved for apps running in **foreground** and require **real time** location updates (e.g. Mapping app).

# Frequency

- In **Request** to Fused Location API for location updates configure `setinterval()` with interval at which location is computed for your app.

# Frequency

- In **Request** to Fused Location API for location updates configure `setinterval()` with interval at which location is computed for your app.
- Use **largest** possible value for the interval.

# Frequency

- In **Request** to Fused Location API for location updates configure `setinterval()` with interval at which location is computed for your app.
- Use **largest** possible value for the interval.
- Especially true for **background** location gathering, often a source of unwelcome battery drain.

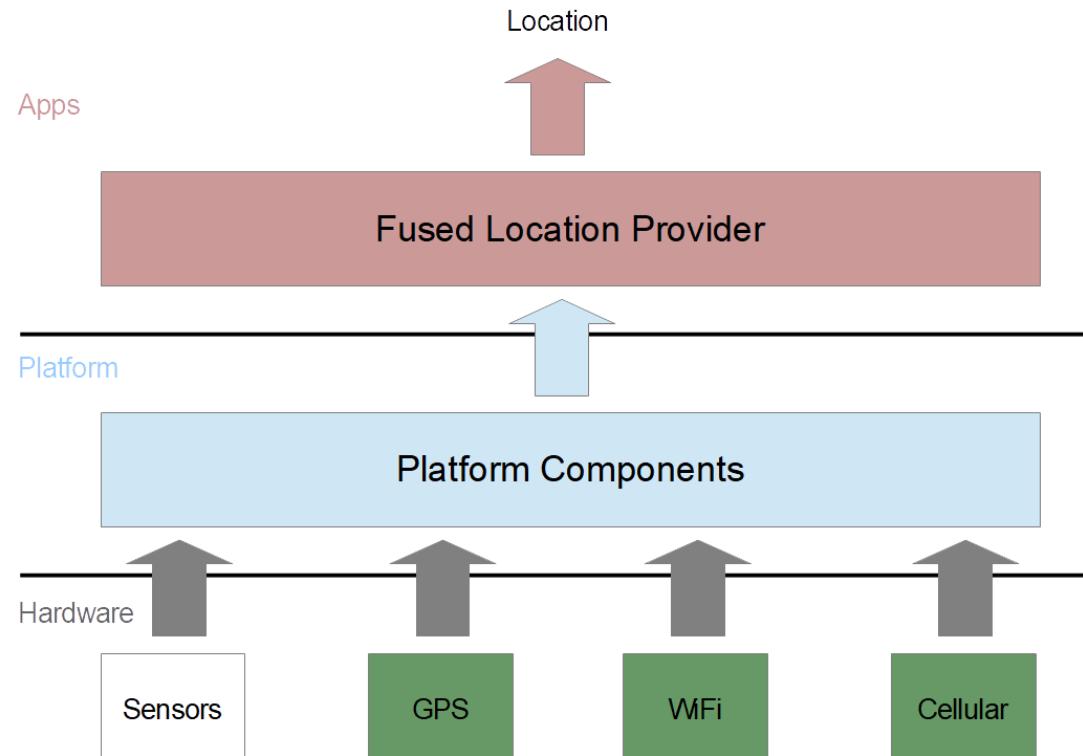
# Frequency

- In **Request** to Fused Location API for location updates configure `setinterval()` with interval at which location is computed for your app.
- Use **largest** possible value for the interval.
- Especially true for **background** location gathering, often a source of unwelcome battery drain.
- Intervals of a few seconds should be **reserved** for foreground use cases.

# Delay

- In **Request** to Fused Location API for location updates use `setMaxWaitTime()`
- Pass a value several times **larger** than interval used in `setInterval()`
- Locations determined at `setInterval()` rate, but delivered in **batch** after interval set in this method.

# Location Updates



LocationRequest

```
setPriority()  
setInterval()  
setMaxWaitTime()
```

Accuracy, Frequency, Delay Settings

1.) Create Request to store details of location updates.

LocationCallback

onLocationResult()

LocationRequest

setPriority()

setInterval()

setMaxWaitTime()

Accuracy, Frequency, Delay Settings

2.) Implement callback for action  
when location updates arrive.

## LocationServices

FusedLocationProviderApi

GeofencingApi

SettingsApi

getFusedLocationProviderClient()

## LocationCallback

onLocationResult()

## LocationRequest

setPriority()

setInterval()

setMaxWaitTime()

Accuracy, Frequency, Delay Settings

### 3.) Get hold of Location Services.

## LocationServices

FusedLocationProviderApi

GeofencingApi

SettingsApi

getFusedLocationProviderClient()

LocationCallback

onLocationResult()

LocationRequest

setPriority()

setInterval()

setMaxWaitTime()

Accuracy, Frequency, Delay Settings

FusedLocationProviderClient

requestLocationUpdates()

removeLocationUpdates()

4.) Ask it for Fused Location Provider Client.

## LocationServices

FusedLocationProviderApi

GeofencingApi

SettingsApi

getFusedLocationProviderClient()

LocationCallback

onLocationResult()

FusedLocationProviderClient

requestLocationUpdates()

removeLocationUpdates()

LocationRequest

setPriority()

setInterval()

setMaxWaitTime()

Accuracy, Frequency, Delay Settings

5.) Request Location Updates with Request and Callback as arguments.

Location

```
double latitude  
double longitude  
double altitude
```

6.) Hardware feeds location data into system and Location object created.

Location

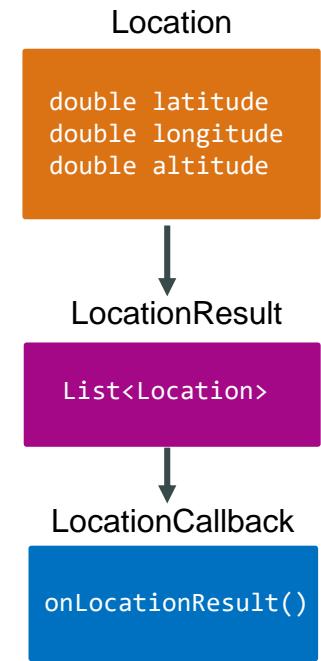
```
double latitude  
double longitude  
double altitude
```



LocationResult

```
List<Location>
```

7.) Gets stored in list with Locations recorded at other times.



8.) Based on delay settings callback called by system with Location Result.

Location

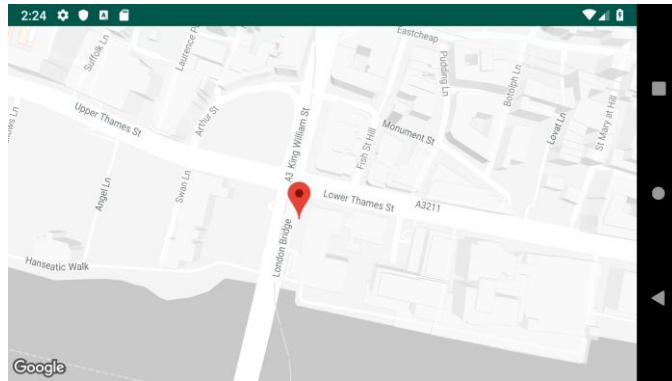
double latitude  
double longitude  
double altitude

LocationResult

List<Location>

LocationCallback

onLocationResult()



9.) Perform action e.g. draw marker  
on a map

FusedLocationProviderClient

```
removeLocationUpdates()
```

10.) When no longer needed,  
best practice to remove updates.

# Best Practices

## Remove Location Updates

- Common source of unnecessary battery drain is failure to remove location updates when **no longer** needed.
- Can happen if activity's onStart() or onResume() methods request location updates **without** call to remove them in onStop() or onPause()

# Best Practices

## Set Timeouts

- Set reasonable timeout when location updates should **stop**.
- Timeout ensures updates don't continue **indefinitely** e.g. in scenarios where updates requested but not removed (due to a bug).
- In Request, add timeout using `setExpirationDuration()`, which takes value for time in milliseconds **since** method was last called.

# Best Practices

## Batch Requests

- For non-foreground use cases, batch multiple requests together.
- Use `setInterval()` to specify interval at which location be computed.
- Then, `setMaxWaitTime()` to set interval at which location is **delivered** to app.
- Value passed to `setMaxWaitTime()` multiple of that for `setInterval()`

# Best Practices

```
LocationRequest request = new LocationRequest();
request.setInterval(10 * 60 * 1000);
request.setMaxWaitTime(60 * 60 * 1000);
```

- Location computed roughly every 10 mins. and 6 location data points delivered in batch per hour.
- Still get location updates every 10 minutes, battery conserved because device woken up only once an hour.

# Permissions

Apps must request location permission in Manifest:

```
<uses-permission  
        android:name = "android.permission.ACCESS_FINE_LOCATION"  
    />
```

ACCESS\_COARSE\_LOCATION

for location accurate to within city block

ACCESS\_FINE\_LOCATION

for precise location

# Permissions

From Marshmallow onwards:

- Users grant or deny **access** to their location for each app.
- Users can change access permission at any time.
- Your app can **prompt** user to grant permission to use location.



Allow **Getting Location Updates** to access this device's location?

---

[Allow all the time](#)

---

[Allow only while using the app](#)

---

[Deny](#)

# Permissions

- User can revoke permission at **any time**.
- Check for permission **each time** your app uses location

# Mobile Software Development

**CMPU 3036 DT211C/3, DT282/3**

# Today's Lecture



Volley

A convenient library for working with  
Network Resources

# Overview

- Volley is HTTP library that makes networking for Android apps easier and most importantly **faster**. It offers following benefits:

# Overview

- Volley is HTTP library that makes networking for Android apps easier and most importantly **faster**. It offers following benefits:
  1. Automatic **scheduling** of network requests.

# Overview

- Volley is HTTP library that makes networking for Android apps easier and most importantly **faster**. It offers following benefits:
  1. Automatic **scheduling** of network requests.
  2. Multiple **concurrent** network connections.

# Overview

- Volley is HTTP library that makes networking for Android apps easier and most importantly **faster**. It offers following benefits:
  1. Automatic **scheduling** of network requests.
  2. Multiple **concurrent** network connections.
  3. Transparent disk and memory response caching.

# Overview

- Volley is HTTP library that makes networking for Android apps easier and most importantly **faster**. It offers following benefits:
  1. Automatic **scheduling** of network requests.
  2. Multiple **concurrent** network connections.
  3. Transparent disk and memory response caching.
  4. Support for request **prioritisation**.

# Overview

- Volley is HTTP library that makes networking for Android apps easier and most importantly **faster**. It offers following benefits:
  1. Automatic **scheduling** of network requests.
  2. Multiple **concurrent** network connections.
  3. Transparent disk and memory response caching.
  4. Support for request **prioritisation**.
  5. **Cancellation** request API for single or multiple requests.

# Overview

- Volley excels at operations used to populate UI e.g. fetching a page of search results as structured data.

# Overview

- Volley excels at operations used to populate UI e.g. fetching a page of search results as structured data.
- Integrates easily with any protocol and comes with **built-in support** for Raw Strings, Images and JSON.

# Overview

- Volley excels at operations used to populate UI e.g. fetching a page of search results as structured data.
- Integrates easily with any protocol and comes with **built-in support** for Raw Strings, Images and JSON.
- Frees you from writing **boilerplate** code and allows you to concentrate on logic that is specific to your app.

# Overview

- Volley is not suitable for **large** download or **streaming** operations, since it holds all responses in memory during parsing.

# Overview

- Volley is not suitable for **large** download or **streaming** operations, since it holds all responses in memory during parsing.
- For large download operations, consider using an alternative like DownloadManager.

# Set Up

- The easiest way to include Volley in your project is to add following **dependency** to app's build.gradle file:

```
dependencies {  
    ...  
    implementation 'com.android.volley:volley:1.1.1'  
}
```

# Set Up

- The easiest way to include Volley in your project is to add following **dependency** to app's build.gradle file:

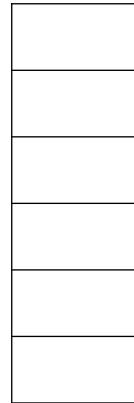
```
dependencies {  
    ...  
    implementation 'com.android.volley:volley:1.1.1'  
}
```

- Also include the following **permission** in app's manifest:

```
android.permission.INTERNET
```

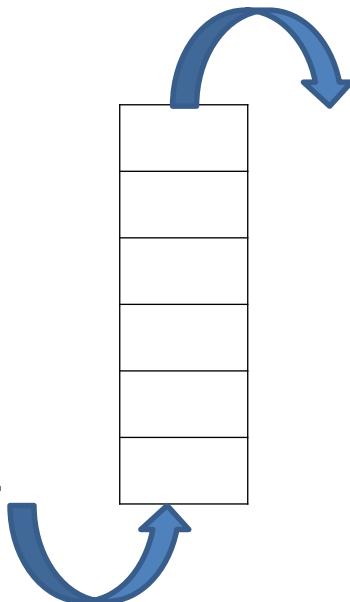
# Sending a Request

- At a **high level** you use Volley by creating:
- A RequestQueue



# Sending a Request

- At a **high level** you use Volley by creating:
- A RequestQueue
- Passing it Request objects.



# Sending a Request

- The RequestQueue manages Worker **Threads** for:

# Sending a Request

- The RequestQueue manages Worker **Threads** for:
  1. Running network operations

# Sending a Request

- The RequestQueue manages Worker **Threads** for:
  1. Running network operations
  2. Reading from and writing to cache

# Sending a Request

- The RequestQueue manages Worker **Threads** for:
  1. Running network operations
  2. Reading from and writing to cache
  3. Parsing responses.

# Sending a Request

- The RequestQueue manages Worker **Threads** for:
  1. Running network operations
  2. Reading from and writing to cache
  3. Parsing responses.
- Requests do **parsing** of raw responses

# Sending a Request

- The RequestQueue manages Worker **Threads** for:
  1. Running network operations
  2. Reading from and writing to cache
  3. Parsing responses.
- Requests do **parsing** of raw responses
- Volley takes care of dispatching parsed response back to **Main Thread** for delivery.

# Sending a Request

- To set up and start a **RequestQueue** with default values use `Volley.newRequestQueue`:

```
final TextView mTextView = (TextView) findViewById(R.id.text);
String url = "https://www.google.com";
RequestQueue queue = Volley.newRequestQueue(this);
```

# Create the Request

```
StringRequest stringRequest = new StringRequest(Request.Method.GET, url,  
        new Response.Listener<String>() {  
            public void onResponse(String response) { ←  
                mTextView.setText("Response is: " +  
                    response.substring(0, 500));  
            },  
            new Response.ErrorListener() {  
                public void onErrorResponse(VolleyError error) {  
                    Log.d(MYTAG, "Some kind of Error");  
                }  
            );
```

Developer writes  
code for listeners  
called in event of  
Success or failure  
of network operation

# Delivery

- Volley always delivers parsed responses on **Main Thread**.

# Delivery

- Volley always delivers parsed responses on **Main Thread**.
- Running on Main Thread convenient for populating UI controls with received data.

# Delivery

- Volley always delivers parsed responses on **Main Thread**.
- Running on Main Thread convenient for populating UI controls with received data.
- As you can freely modify UI controls **directly from** your response handler

```
public void onResponse(String response) {  
    mTextView.setText("Response is: " + response.substring(0, 500));  
}
```

# Add it to the Queue

- After constructing a Request, to send it, add it to queue

```
queue.add(stringRequest);
```

# Add it to the Queue

- After constructing a Request, to send it, add it to queue

```
queue.add(stringRequest);
```

- Once you add Request it:
  1. Moves through the **Pipeline**,
  2. Gets **Serviced**
  3. Raw Response Parsed
  4. **Delivered**

# Add it to the Queue

- When you call add(), Volley runs:
  1. One **Cache** Processing Thread
  2. A **Pool** of Network Dispatch Threads (Default 4)

# Add it to the Queue

- When you call add(), Volley runs:
  1. One **Cache** Processing Thread
  2. A **Pool** of Network Dispatch Threads (Default 4)
- When Request added to queue, picked up by Cache Thread and **triaged**:

# Add it to the Queue

- When you call add(), Volley runs:
  1. One **Cache** Processing Thread
  2. A **Pool** of Network Dispatch Threads (Default 4)
- When Request added to queue, picked up by Cache Thread and **triaged**:
- If Request can be serviced from cache, cached response is parsed on Cache Thread and parsed response delivered on Main Thread.

# Add it to the Queue

- If Request cannot be serviced from cache, placed on network **queue.**

# Add it to the Queue

- If Request cannot be serviced from cache, placed on network **queue**.
- 1<sup>st</sup> available Network Thread takes Request from queue:
  1. Performs HTTP transaction.

# Add it to the Queue

- If Request cannot be serviced from cache, placed on network **queue**.
- 1<sup>st</sup> available Network Thread takes Request from queue:
  1. Performs HTTP transaction.
  2. Parses Response on worker thread.

# Add it to the Queue

- If Request cannot be serviced from cache, placed on network **queue**.
- 1<sup>st</sup> available Network Thread takes Request from queue:
  1. Performs HTTP transaction.
  2. Parses Response on worker thread.
  3. Writes Response to cache.

# Add it to the Queue

- If Request cannot be serviced from cache, placed on network **queue**.
- 1<sup>st</sup> available Network Thread takes Request from queue:
  1. Performs HTTP transaction.
  2. Parses Response on worker thread.
  3. Writes Response to cache.
  4. Posts parsed Response back to Main Thread for delivery.

# Add it to the Queue

- Note: that expensive operations like **blocking** I/O and parsing/decoding done on Worker Threads.

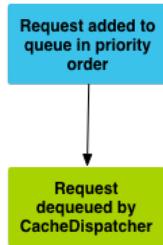
# Add it to the Queue

- Note: that expensive operations like **blocking** I/O and parsing/decoding done on Worker Threads.
- You can add Request from any thread, but responses are always delivered on Main Thread.

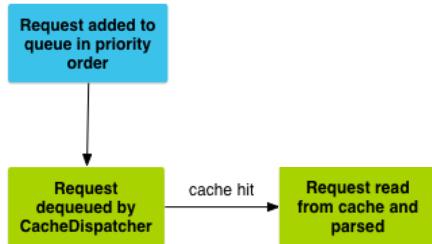
# How it Works

Request added to  
queue in priority  
order

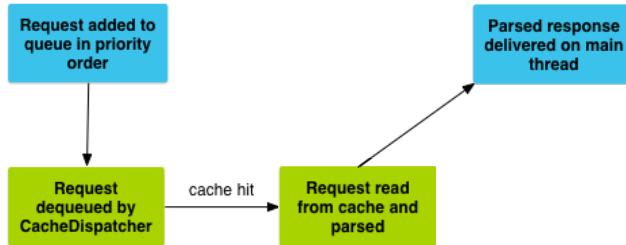
# How it Works



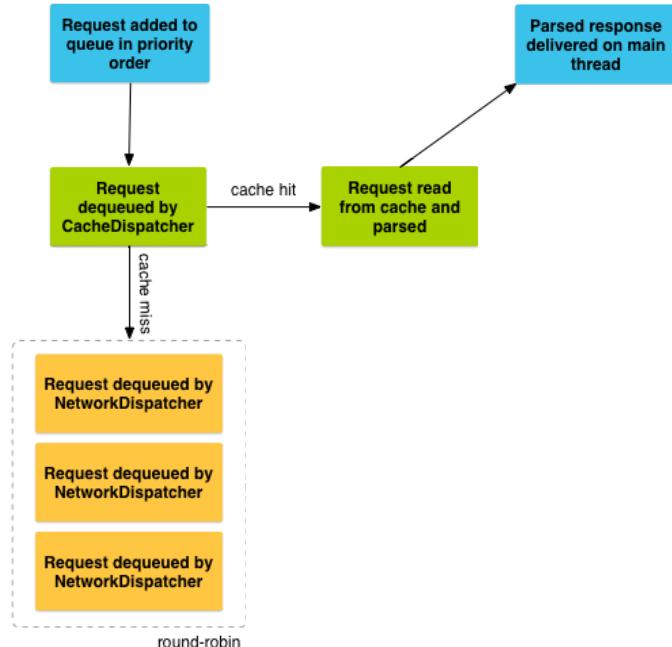
# How it Works



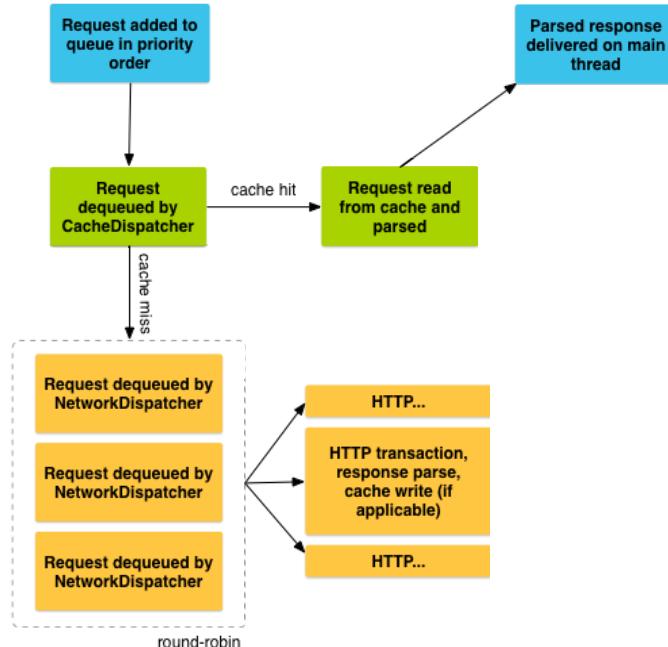
# How it Works



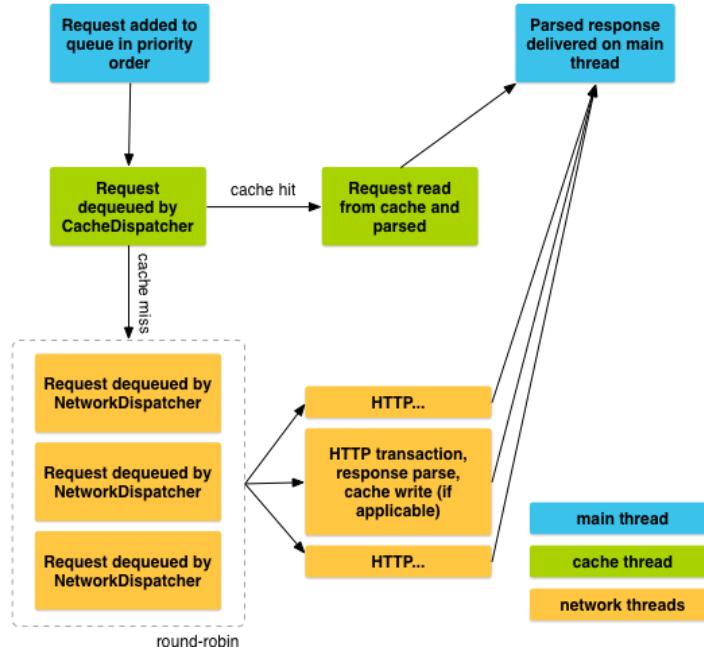
# How it Works



# How it Works



# How it Works



# Request Class

- Request class is **abstract** and contains the following two methods:

```
protected abstract Response<T> parseNetworkResponse(NetworkResponse response);
```

is responsible for **parsing** the Network Response

# Request Class

- Request class is **abstract** and contains the following two methods:

```
protected abstract Response<T> parseNetworkResponse(NetworkResponse response);
```

is responsible for **parsing** the Network Response

```
protected abstract void deliverResponse(T response);
```

then **delivers** the resulting response

# Examples

- `StringRequest` subclasses `Request` and `Volley` provides an **implementation** for these two methods (overrides):

```
@Override  
protected Response<String> parseNetworkResponse(NetworkResponse response)  
{  
    String parsed = new String(response.data);  
    return Response.success(parsed);  
}
```

Note: for brevity some of the code is omitted above

# Examples

- StringRequest subclasses Request and Volley provides an **implementation** for these two methods (overrides):

```
@Override  
protected void deliverResponse(String response) {  
    if (listener != null) {  
        listener.onResponse(response);  
    }  
}
```

Note: the listener itself is defined by the developer.

# Examples

- In this case some textual data lives at the url.



A screenshot of a web browser window. The address bar shows 'view-source:https://www.google.com/'. The page content area displays the first few lines of the HTML source code for Google's homepage, including the doctype declaration and the opening  tag with attributes like 'itemscope' and 'itemtype'. The browser interface includes standard navigation buttons (back, forward, search) and a toolbar with icons for copy, paste, and download.

```
1 <!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="en-IE"><head><meta charset="UTF-8"><meta content="origin" name="referrer">
```

# Examples

- **StringRequest** parses the response and lets us display the text on screen:



Response is: <!doctype html><html lang="en-IE"><head><meta charset="UTF-8"><meta content="width=device-width,minimum-scale=1.0" name="viewport"><meta content="telephone=no" name="format-detection"><meta content="address=no" name="format-detection"><meta content="/images/branding/googleleg/1x/googleleg\_standard\_color\_128dp.png" itemprop="image"><title>Google</title><script nonce="6CE6psfnre3pWfb/7RNJcA==">(function(){window.google={kEI:'3YgIXOurFNaN1fAPu8eEkA8',authuser:0,kscs:'c9c918f0\_3YgIXOurFNaN1fAPu8eEkA8',q

# Examples

- Same idea for **JsonObjectRequest** i.e. Volley implements the two methods:

```
@Override  
protected Response<JSONObject> parseNetworkResponse(NetworkResponse response)  
{  
    String jsonString = new String(response.data);  
    return Response.success(new JSONObject(jsonString));  
}
```

Note: for brevity some of the code is omitted above

# Examples

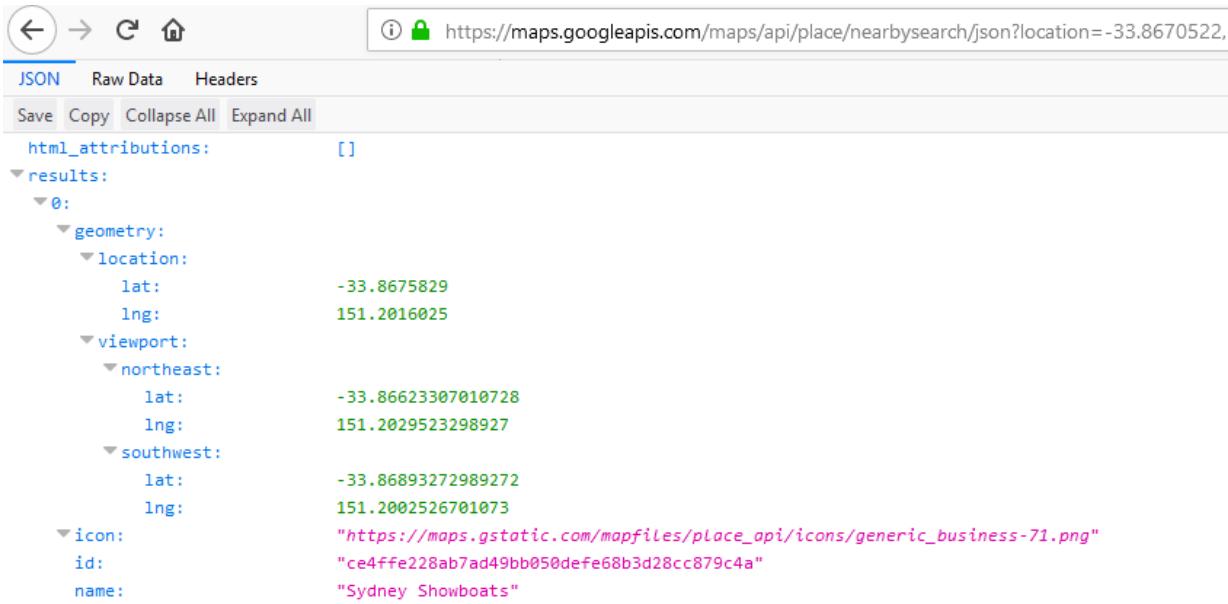
- Same idea for **JsonObjectRequest** i.e. Volley implements the two methods:

```
@Override  
protected void deliverResponse(JSONObject response) {  
    if (listener != null) {  
        listener.onResponse(response);  
    }  
}
```

Note: the listener itself is defined by the developer.

# Examples

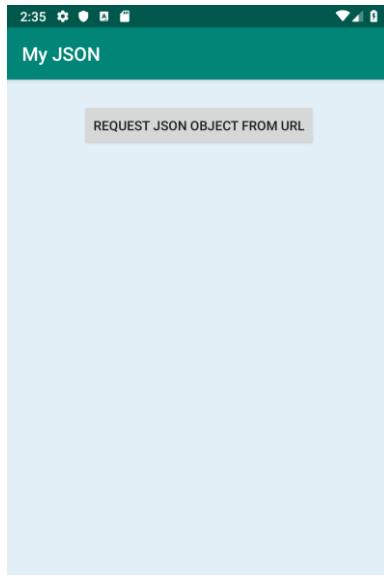
- API call returns data on **restaurants** formatted as JSON



The screenshot shows a browser developer tools Network tab with a JSON response. The URL is https://maps.googleapis.com/maps/api/place/nearbysearch/json?location=-33.8670522,151.2016025. The response is a JSON object with the following structure:

```
html_attributions: []
results:
  0:
    geometry:
      location:
        lat: -33.8675829
        lng: 151.2016025
      viewport:
        northeast:
          lat: -33.86623307010728
          lng: 151.2029523298927
        southwest:
          lat: -33.86893272989272
          lng: 151.2002526701073
    icon: "https://maps.gstatic.com/mapfiles/place_api/icons/generic_business-71.png"
    id: "ce4ffe228ab7ad49bb050defe68b3d28cc879c4a"
    name: "Sydney Showboats"
```

# Examples



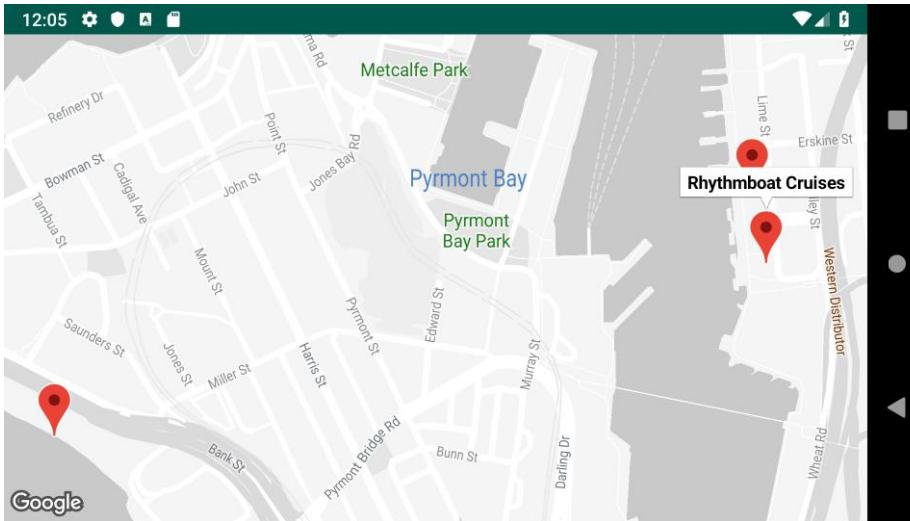
# Examples

- `JsonObjectRequest` parses JSON response and lets us extract values from **Java object**.



# Examples

- Location data can then be **displayed** on a Map.



# Examples

- There is a little bit more **processing** for an **ImageRequest**:

```
@Override  
protected Response<Bitmap> parseNetworkResponse(NetworkResponse response)  
{  
    byte[] data = response.data;  
    Bitmap bitmap = BitmapFactory.decodeByteArray(data, data.length);  
    return Response.success(bitmap);  
}
```

Note: for brevity some of the code is omitted above

# Examples

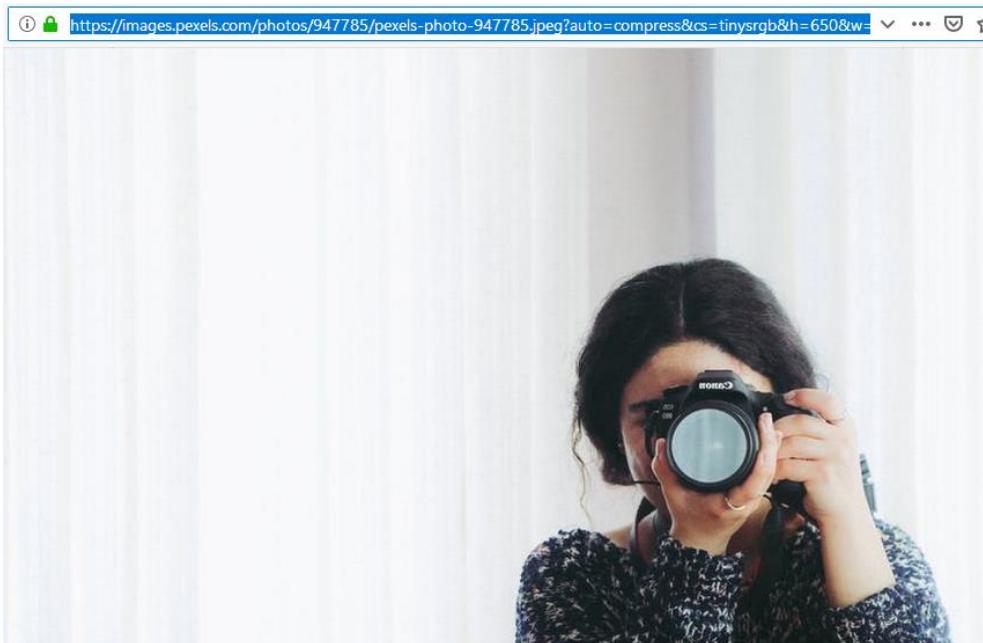
- There is a little bit more **processing** for an **ImageRequest**:

```
@Override  
protected void deliverResponse(Bitmap response) {  
    if (listener != null) {  
        listener.onResponse(response);  
    }  
}
```

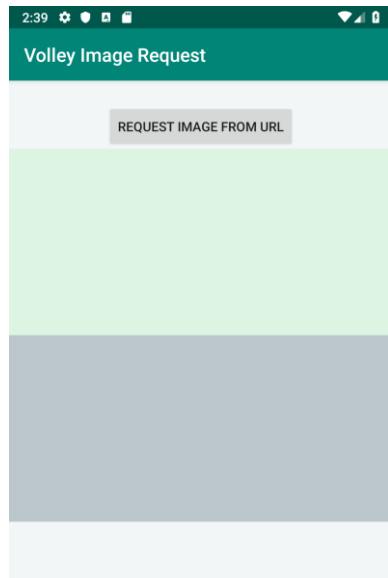
Note: the listener itself is defined by the developer.

# Examples

- In this case we want to download an **image** from a url.

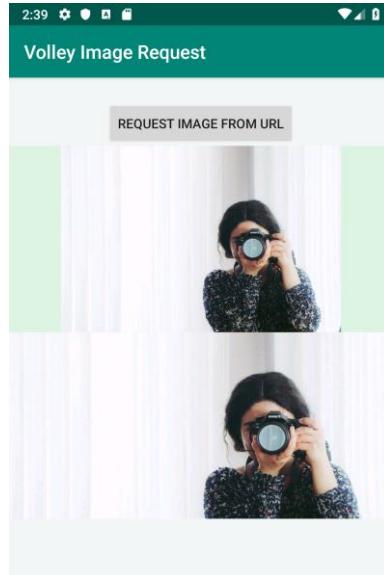


# Examples



# Examples

- `ImageRequest` decodes the response and delivers the **Bitmap** for display



# Cancel a Request

- To cancel a request, call `cancel()` on your `Request` object.

# Cancel a Request

- To cancel a request, call `cancel()` on your `Request` object.
- Once cancelled, Volley guarantees response handler will **never** be called.

# Cancel a Request

- To cancel a request, call `cancel()` on your `Request` object.
- Once cancelled, Volley guarantees response handler will **never** be called.
- The `onStop()` method is a place to cancel any **pending** Requests.

# Cancel a Request

- A **tag** can also be associated with each Request.

# Cancel a Request

- A **tag** can also be associated with each Request.
- The tag can then be used to provide a scope of requests to cancel.

```
public static final String TAG = "MyTag";  
...  
stringRequest.setTag(TAG);  
mRequestQueue.add(stringRequest);
```

# Cancel a Request

- Then in Activity's `onStop()` method, cancel all requests that have this tag:

```
@Override  
protected void onStop () {  
    super.onStop();  
    if (mRequestQueue != null) {  
        mRequestQueue.cancelAll(TAG);  
    }  
}
```

# Mobile Software Development

**CMPU 3036 DT211C/3, DT282/3**

# Today's Topics

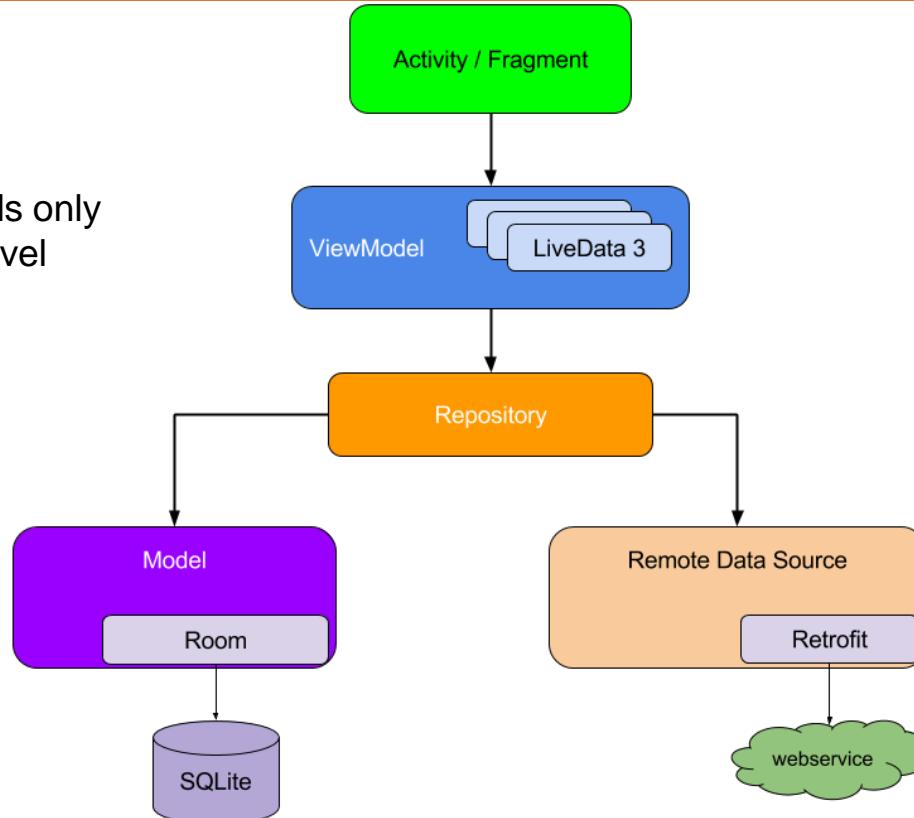
- Architecture Components
- Exam Brief

# Problem - Bloat

- It's a common mistake to write all your code in an Activity or a Fragment – A God Class.
- These UI-based classes should only contain logic that handles UI and operating system interactions.

# Solution - Separation of Concerns

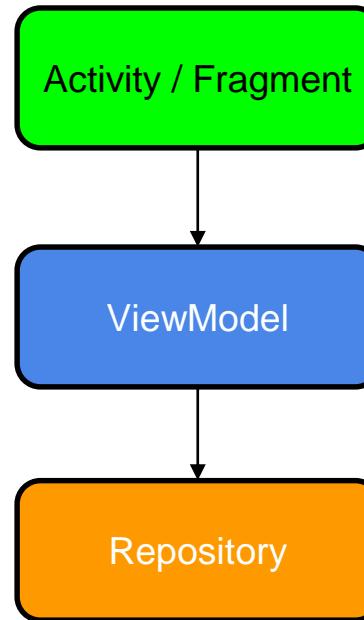
Each module depends only  
on the module one level  
below it.



# ViewModel

A ViewModel object:

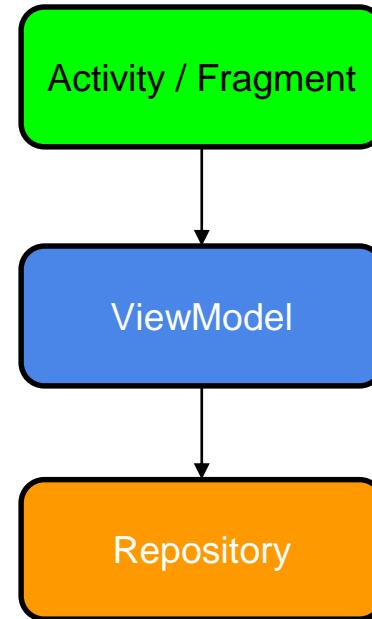
1. Provides the **data** for a specific UI component, such as a fragment or activity.
2. Contains data-handling business logic to communicate with the model.



# ViewModel

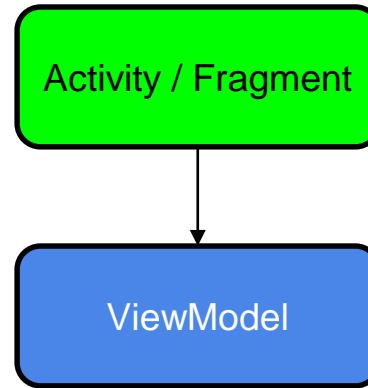
For example, the ViewModel can:

1. Call other components to load the data.
2. Forward user requests to modify the data.

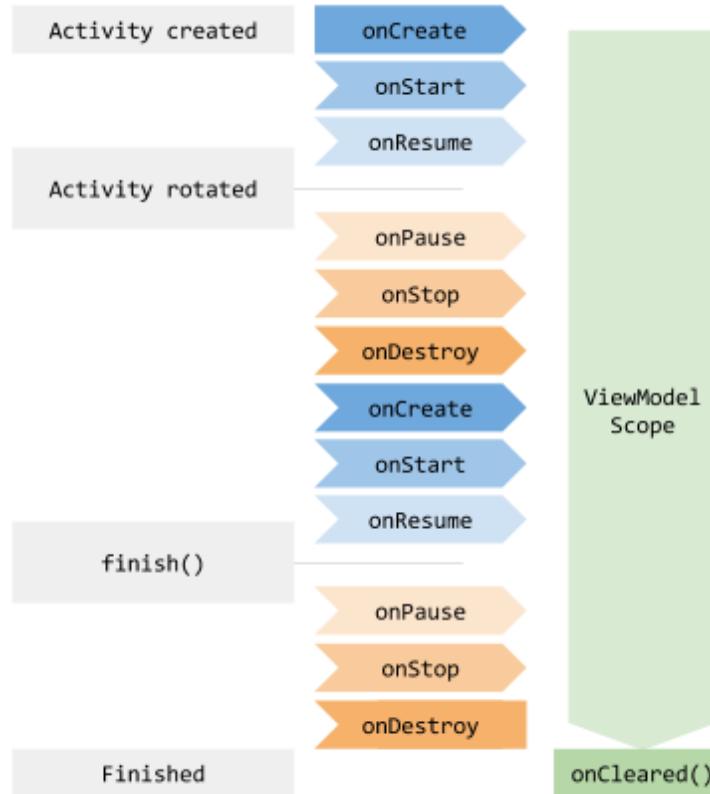


# ViewModel

- The ViewModel doesn't know about UI components.
- So, it isn't affected by configuration changes, such as recreating an activity when rotating the device.

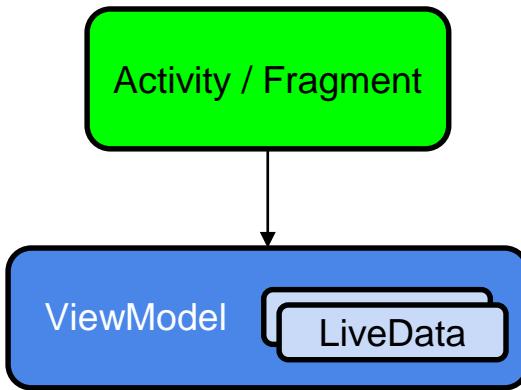


# ViewModel



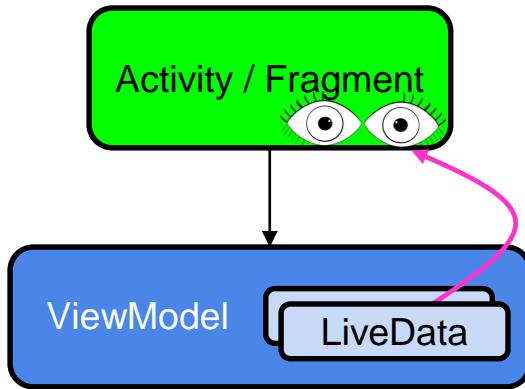
# LiveData

- LiveData is an **observable** data holder class.



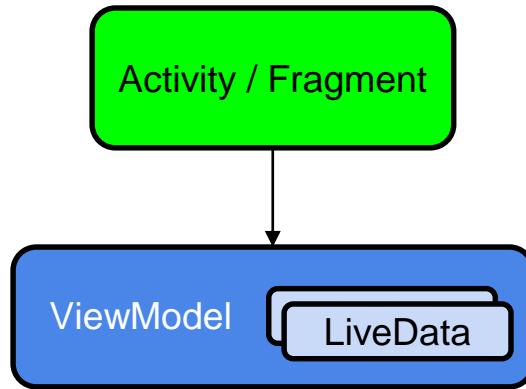
# LiveData

- LiveData is an **observable** data holder class.
- This simply means that Activities who **observe** the LiveData are **notified** when the data changes.

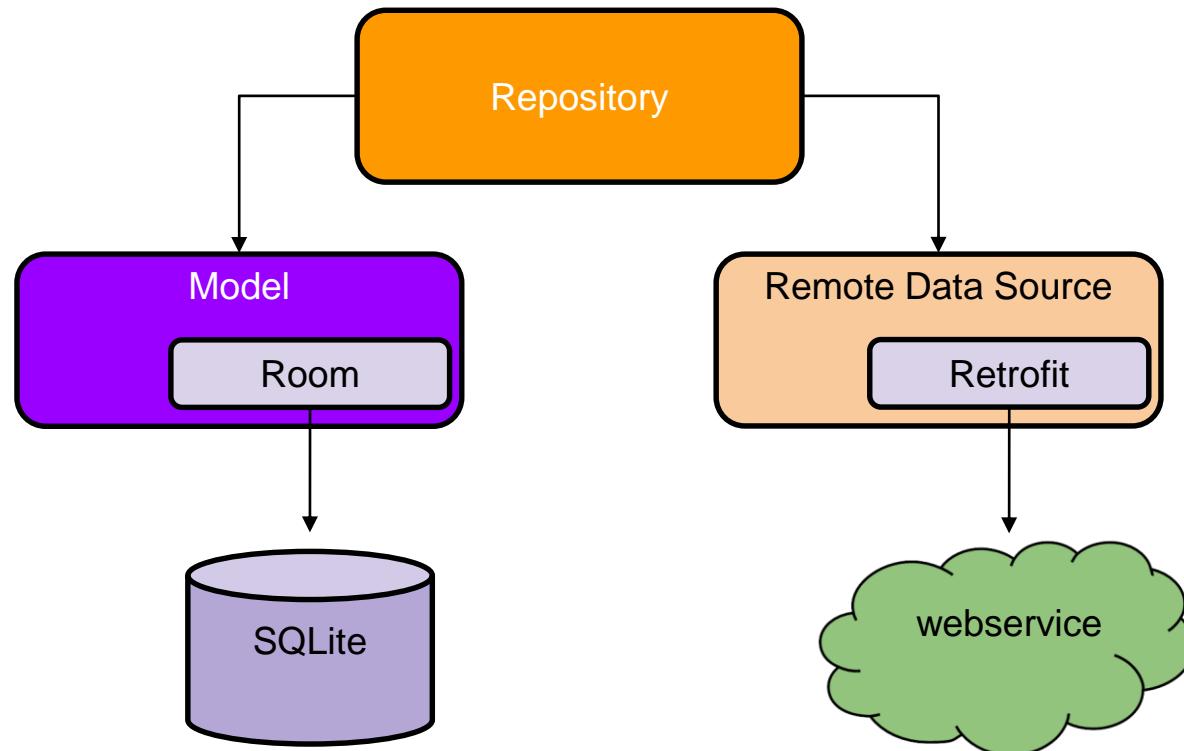


# LiveData

- LiveData is also **lifecycle-aware**, meaning it respects the lifecycle of other app components e.g. activities, fragments etc..
- This awareness ensures LiveData only updates app component observers that are in an active lifecycle state.



# Repository



# Repository

- Repository modules handle data operations.
- They provide a clean API so that the rest of the app can retrieve this data easily.
- They know where to get the data from and what API calls to make when data is updated.

# Repository

- You can consider repositories to be mediators between different data sources, such as persistent models, web services, and caches.

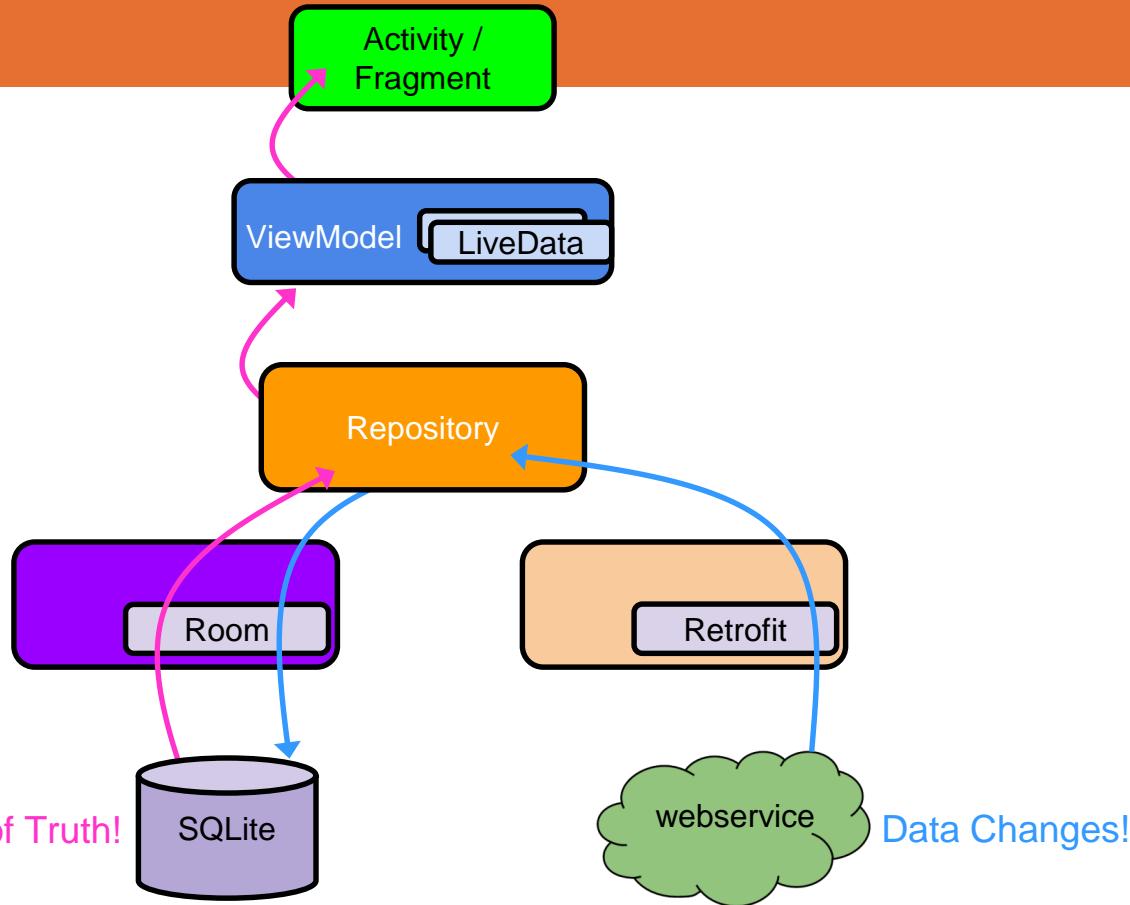
# Room

- Room is an object-mapping library that provides local data persistence with minimal boilerplate code.
- At compile time, it validates each query against your data schema, so broken SQL queries result in compile-time errors instead of runtime failures.

# Room

- Room abstracts away some of the underlying implementation details of working with raw SQL tables and queries.
- It also allows you to observe changes to the database's data, including collections and join queries, exposing such changes using **LiveData** objects.

# Typical Pattern



# Final Notes

- This design creates a consistent and pleasant user experience.
- Separation of concerns also facilitates easy testability and maintenance of each code module.

# Learn More...

- There is a really nice walk through of how the main components of the architecture can be put into practice to create an app at [this link!](#)
- If you get the hang of this tutorial then, in my opinion, you have successfully completed the module :D