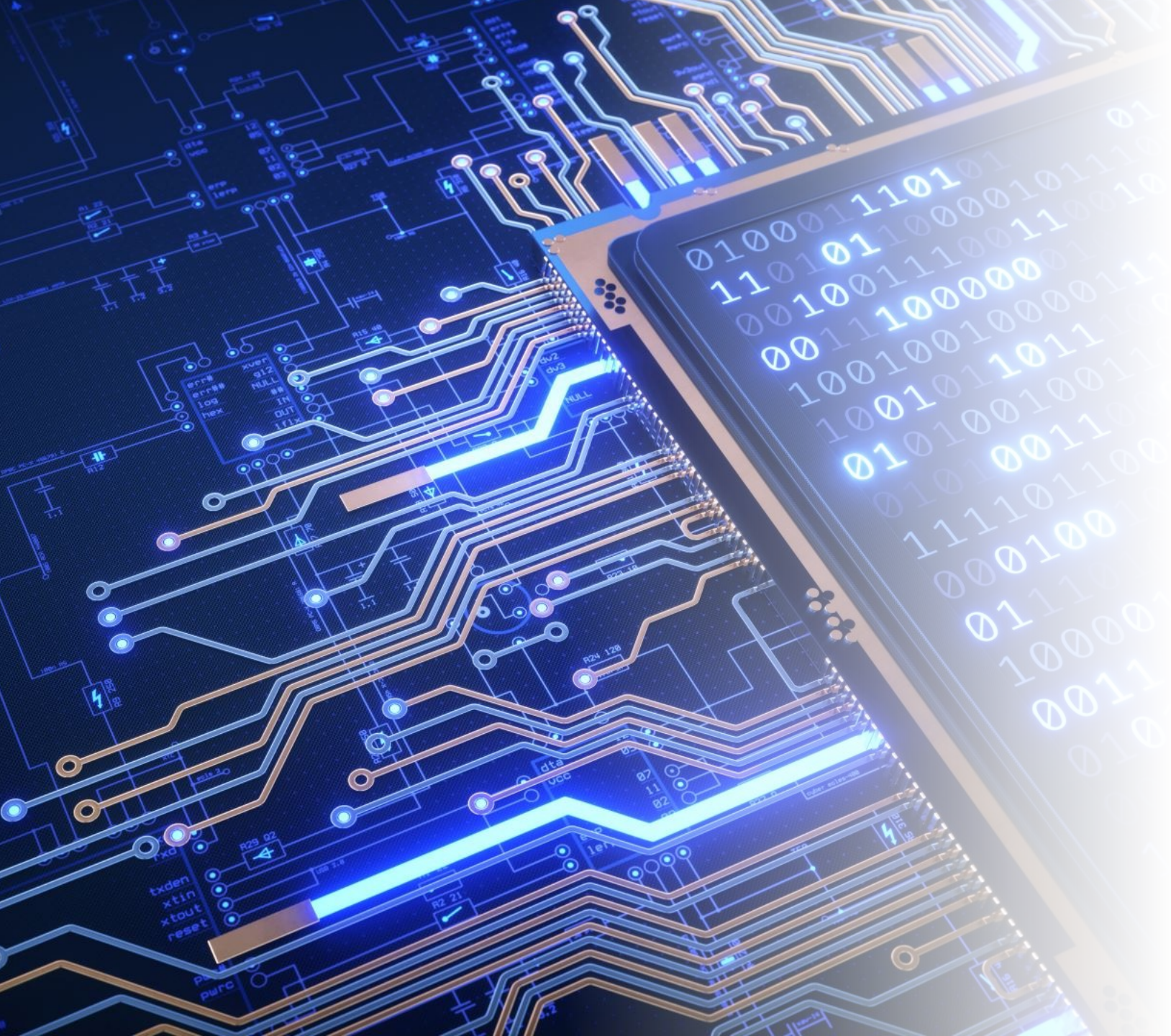


# EEP 523: MOBILE APPLICATIONS FOR SENSING AND CONTROL

SUMMER 2021
Lecture 4

Tamara Bonaci

tbonaci@uw.edu

# Agenda

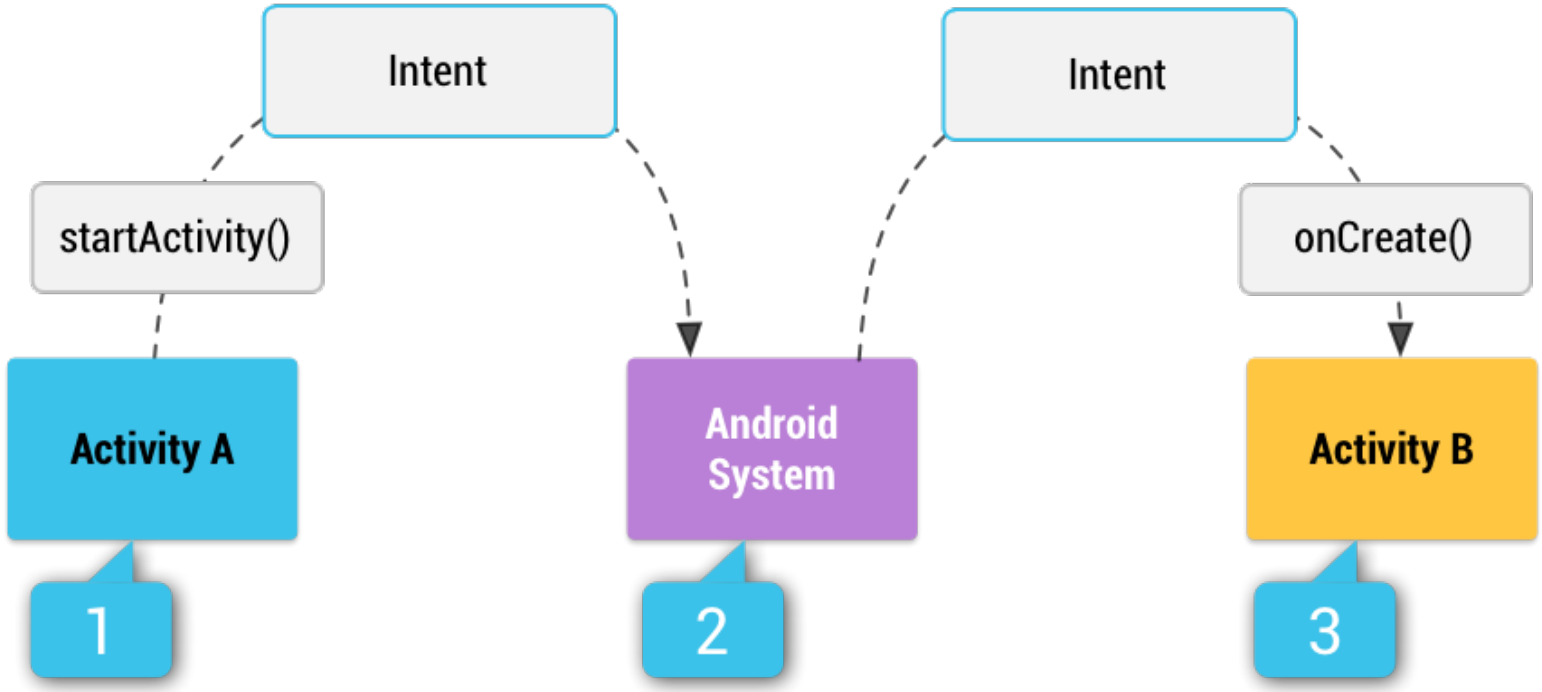
Review: Intents
- Implicit Intent – taking a picture
- Bitmaps and Drawables
- 2D Graphics
- Google Mobile Vision API
- Fragments
- Android Sensors
  - Accelerometer
  - Gyroscope
  - Composite sensors
- Libraries in Android

# Review: Intents

An ***Intent*** is a messaging object you can use to request an action from another app component.

- **Explicit intents**
  - Specify which application will satisfy the intent
  - You'll typically use an explicit intent to start a component in your own app, because you know the class name of the activity or service you want to start
- **Implicit intents**
  - Do not name a specific component, but instead declare a general action to perform, which allows a component from another app to handle it.
  - Example: show the user a location on a map: use an implicit intent to request that another capable app show a specified location on a map.

# Review: Implicit Intent

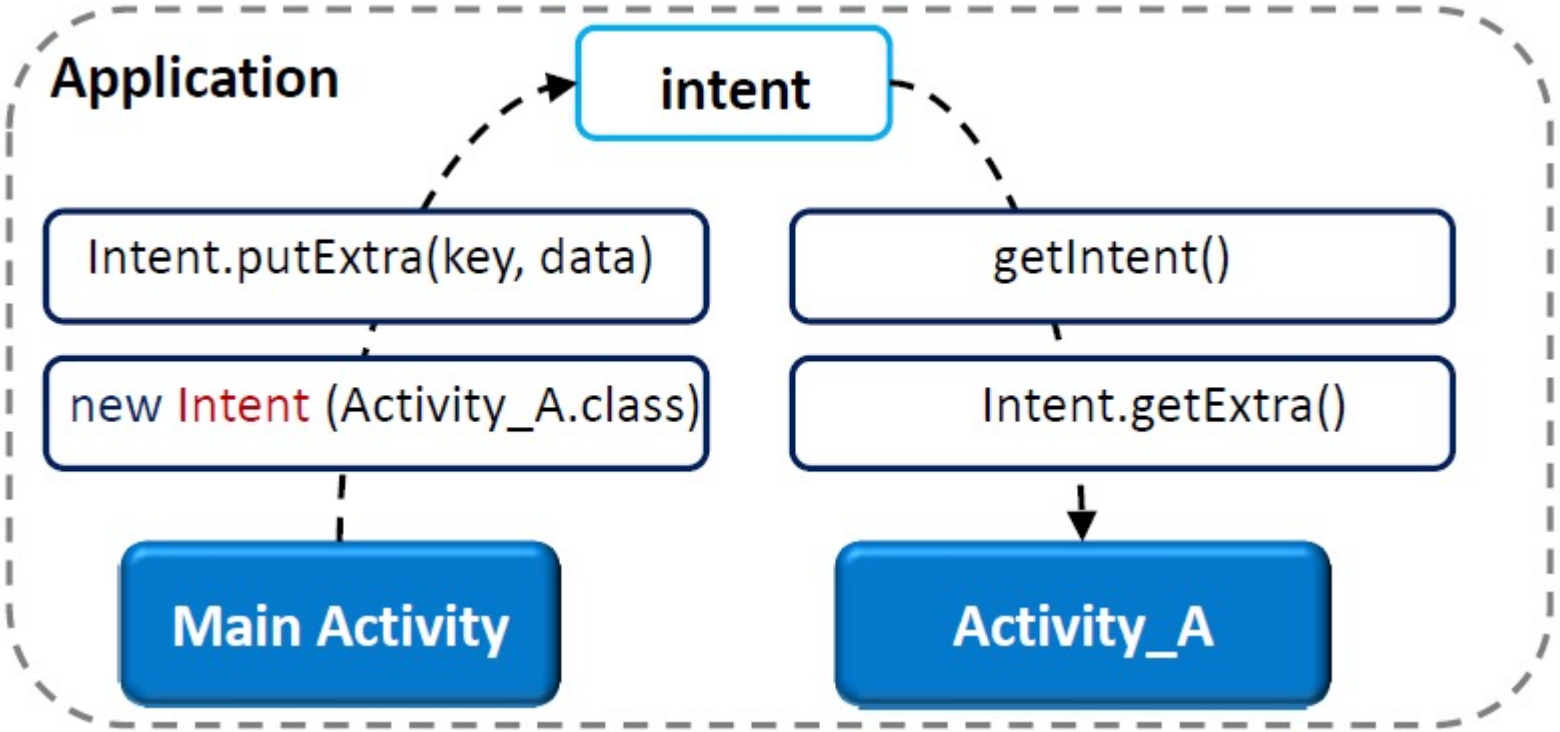



*Activity A* creates an Intent with an action description and passes it to startActivity().

The Android System searches all apps for an **intent filter** that matches the intent.

Match found: system starts the matching activity (Activity B) by invoking its onCreate() method and passing it the Intent.

# Review: Explicit Intent




Intent.putExtra(…) comes in many flavors, but it always has two arguments.

- key: always a String key,
- data: type will vary. It returns the Intent itself, so you can chain multiple calls if you need to.

# Review: Passing Data to a Second Activity

*MainActivity.kt*


```
const val KEY = “myMessage" //any String that you define

fun sendMessage(view: View) {
        val editText = findViewById<EditText>(R.id.editText)
        val message = editText.text.toString()
        val intent = Intent(this, SecondActivity::class.java)
        intent.putExtra(KEY, message)

        startActivity(intent)
}
```

*2. Put data into the intent*

*3. Launch the intent*

# Review: Passing Multiple Data Points to a Second Activity

```
const val KEY = “myMessage" //any String that you define

fun sendMessage(view: View) {
        val editText = findViewById<EditText>(R.id.editText)
        val message = editText.text.toString()
        val intent = Intent(this, SecondActivity::class.java)
        intent.putExtra(KEY1, message1)
        intent.putExtra(KEY1, message2)
    intent.putExtra(KEY3, message3)


        startActivity(intent)

}
```

# Passing Data in Between Activities

***Extras***

- arbitrary data that the calling activity can include with an intent.

- You can think of them like constructor arguments, even though you cannot use a custom constructor with an activity subclass.

- The OS forwards the intent to the recipient activity, which can then access the extras and retrieve the data,

```
Intent.putExtra(…)
```

# Summary: *Explicit* Intents

## Passing multiple Basic Types

a) Multiple calls to putExtra

```
In your ActivityA.kt
val i = Intent(this, KnotesDetails::class.java).putExtra("KEY1", value1)
i.putExtra("KEY2",value2)
i.putExtra("KEY3, value3)
   ...
And then in your ActivityB.kt
val id = intent.getIntExtra("KEY1",default)
val id2 = intent.getIntExtra("KEY2",default)
val id3 = intent.getIntExtra("KEY3",default)
```

b) Passing an array

```
In your ActivityA.kt
private val COUNTRIES = arrayOf("Australia", "Brazil")
val i = Intent(this, ActivityB::class.java).putExtra("KEY", COUNTRIES)

In your ActivityB.kt
val country_list= intent.getStringArrayExtra("KEY",default)
country_first= country_list[0]
```

c) Passing an object

# *Summary: Explicit* intents : Passing Objects

## 1. Create class that extends ***Parcelable***

```
import android.os.Parcelable
import kotlinx.android.parcel.Parcelize

@Parcelize class Book(val title: String, val author: String, val year: Int) : Parcelable
```

## 2. Passing the *Parcelable*

```
val myBook = Book(“myTitle”, ”myAuthor”, 2019)

val intent = Intent(this, AnotherActivity::class.java)
intent.putExtra("extra_item", myBook)
```

## 3. Retrieving the Parcelable

```
val item = intent.getParcelableExtra<Book>("extra_item")
// Do something with the item (example: set Item title and price)
val receivedTitle = item.title
```

# Summary: Intents: Implicit or Explicit ?

**Explicit**: Start known/explicit Activity

```
val i = Intent(this, KnotesDetails::class.java)
i.putExtra("KEY", mKnoteID)
startActivityForResult(i, PICK_KNOTE_REQUEST)
```

This (current activity) ➡ KnotesDetails activity

---

**Implicit**: Start an action

```
 Intent(MediaStore.ACTION_IMAGE_CAPTURE).also { takePictureIntent ->
        takePictureIntent.resolveActivity(packageManager)?.also {
            startActivityForResult(takePictureIntent,
REQUEST_IMAGE_CAPTURE)
```

Ccurrent activity ➡ ACTION_IMAGE_CAPTURE

# *Summary: Implicit* intents

- An intent allows you to start an activity in another app by describing a simple action you'd like to perform (such as "view a map" or "take a picture") in an Intent object.

- This type of intent is called an *implicit* intent because it does not specify the app component to start, but instead specifies an ***action*** and provides some *data* with which to perform the action.

- When you call startActivity() or startActivityForResult() and pass it an implicit intent, the systemresolves the intent to an app that can handle the intent and starts its corresponding Activity.

**If there's more than one app that can handle the intent, the system presents the user with a dialog to pick which app to use**.

# Common *Implicit* intents *(link)*

| Main App | Actions |
|---|---|
| **Alarm clock** | https://developer.android.com/guide/components/intents-common#Clock |
| **Calendar** | https://developer.android.com/guide/components/intents-common#Calendar |
| **Camera** | https://developer.android.com/guide/components/intents-common#Camera |
| **Contacts** | https://developer.android.com/guide/components/intents-common#Contacts |
| **Email** | https://developer.android.com/guide/components/intents-common#Email |
| **File Storge** | https://developer.android.com/guide/components/intents-common#Storage |
| **Maps** | https://developer.android.com/guide/components/intents-common#Maps |
| **Music or Video** | https://developer.android.com/guide/components/intents-common#Music |
| **New note** | https://developer.android.com/guide/components/intents-common#NewNote |
| **Phone** | https://developer.android.com/guide/components/intents-common#Phone |
| **Search** | https://developer.android.com/guide/components/intents-common#Search |
| **Settings** | https://developer.android.com/guide/components/intents-common#Settings |
| **Text messaging** | https://developer.android.com/guide/components/intents-common#Messaging |
| **Browser** | https://developer.android.com/guide/components/intents-common#Browser |
| | |

# Implicit Intent: Taking a Picture

EE P 523, Lecture 4

# *Implicit* intents: Taking Pictures

Example App "*TakePicture*" on Canvas

1. **Get permissions**
   a) Manifest

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="edu.uw.eep523.facedetectorpicture">
    <uses-permission android:name="android.permission.CAMERA" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

    <uses-feature android:name="android.hardware.camera" />
    <uses-feature android:name="android.hardware.camera.autofocus" />
```

   b) At runtime

```
getRuntimePermissions()
if (!allPermissionsGranted()) {
    getRuntimePermissions()
}
```

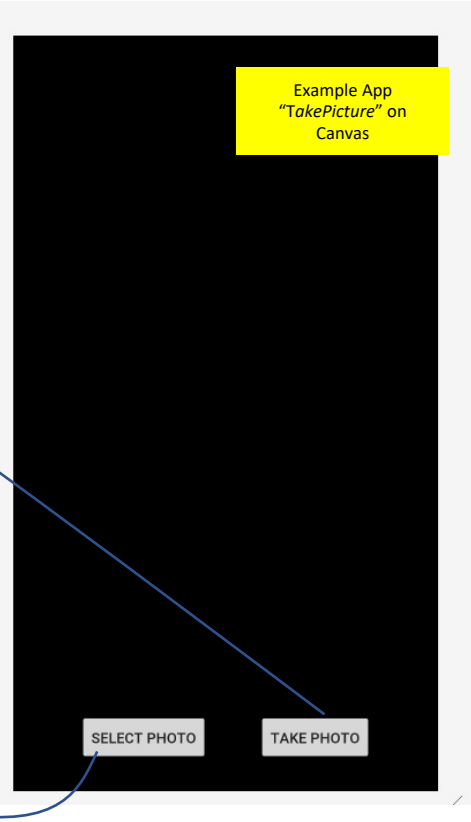
```
private fun getRequiredPermissions(): Array<String?> {
    return try {
        val info = this.packageManager.getPackageInfo(this.packageName, PackageManager.GET_PERMISSIONS)
        val ps = info.requestedPermissions
        if (ps != null && ps.isNotEmpty()) {
            ps
        } else {
            arrayOfNulls(0)
        }
    } catch (e: Exception) {
        arrayOfNulls(0)
    }
}
```

# *Implicit* intents: Taking Pictures

## 2. Launch the intent

```
fun startCameraIntentForResult(view:View) {
    // Clean up last time's image
    imageUri = null
    previewPane?.setImageBitmap(null)

    val takePictureIntent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
    takePictureIntent.resolveActivity(packageManager)?.let {
        val values = ContentValues()
        values.put(MediaStore.Images.Media.TITLE, "New Picture")
        values.put(MediaStore.Images.Media.DESCRIPTION, "From Camera")
        imageUri = contentResolver.insert(MediaStore.Images.Media.EXTERNAL_CONTENT_URI, values)
        takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT, imageUri)
        startActivityForResult(takePictureIntent, REQUEST_IMAGE_CAPTURE)
    }
}
```

Take picture and add to gallery




```
fun startChooseImageIntentForResult(view:View) {
    val intent = Intent()
    intent.type = "image/*"
    intent.action = Intent.ACTION_GET_CONTENT
    startActivityForResult(Intent.createChooser(intent, "Select Picture"), REQUEST_CHOOSE_IMAGE)
}
```

Choose picture from gallery

# *Implicit* intents: Taking Pictures

**3. Recover the image from the intent:** image is recovered as a Bitmap

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode == Activity.RESULT_OK) {
        tryReloadAndDisplayImage()
    } else if (requestCode == REQUEST_CHOOSE_IMAGE && resultCode == Activity.RESULT_OK) {
        // In this case, imageUri is returned by the chooser, save it.
        imageUri = data!!.data
        tryReloadImage()
    }
}
```

```
private fun tryReloadImage() {
        val imageBitmap = if (Build.VERSION.SDK_INT < 29) {
            MediaStore.Images.Media.getBitmap(contentResolver, imageUri)
        } else {
            val source = ImageDecoder.createSource(contentResolver, imageUri!!)
            ImageDecoder.decodeBitmap(source)
        }
    }
}
```

# *Implicit* intents: Take Pictures

**4. Display the Image in a widget/view** *(you will need both for assignments 2 and 3)*

a) In an **ImageView**: `imageViewID?.setImageBitmap(imageBitmap)`

b) In a **DrawView** :

```
drawView?.background= BitmapDrawable(getResources(), imageBitmap);
```

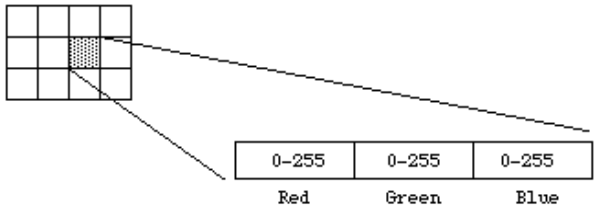
Image from
the camera or gallery

# Bitmaps and Drawables

## Bitmap

Regular rectangular mesh of cells called pixels, each pixel containing a color value. They are characterized by only two parameters, the number of pixels and the information content (color depth) per pixel.

**24 bit RGB**
8 bits allocated to each red, green, and blue component. In each component the value of 0 refers to no contribution of that colour, 255 refers to fully saturated contribution of that colour. Since each component has 256 different states there are a total of 16777216 possible colours.




## Drawable

General abstraction for "something that can be drawn."

- The Drawable class provides a generic API for dealing with an underlying visual resource that may take a variety of forms.
- Unlike a View, a Drawable does not have any facility to receive events or otherwise interact with the user.

# Bitmap

Using a BitmapFactory, you can create bitmaps in 3 common ways:

- from a resource – you can create a bitmap from an image (such as .png or .jpg) located in your drawable folder.

```
val pict = BitmapFactory.decodeResource(resources,R.drawable.bored)
```

- file

- InputStream (such as an image from the camera)

# Drawables (/res)

Supported file types:

- PNG (preferred),
- JPG (acceptable)
- GIF (discouraged).

# Supporting Bitmaps

- Bitmaps can very easily exhaust an app's memory budget.
  - For example, the camera on the Pixel phone takes photos of up to 4048x3036 pixels (12 megapixels).

- If the bitmap configuration used is ARGB_8888, the default for Android 2.3 (API level 9) and higher, loading a single photo into memory takes about 48MB of memory (4048*3036*4 bytes).

- Loading bitmaps on the UI thread can degrade your app's performance, causing slow responsiveness. It is therefore important to manage threading appropriately when working with bitmaps.

# 2D Graphics

EE P 523, Lecture 4

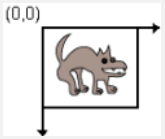
# Drawing 2D Graphics

- To draw our own custom 2D graphics on screen, we'll make a **custom View subclass** with the drawing code.

- If the app is animated (such as a game), we'll also use a **thread** to periodically update the graphics and redraw them.

# Custom View Template

```
class BallView(context: Context, attrs: AttributeSet) : View(context, attrs)
{
    //Define the ball shape
    private val size = 100f
    private var ballX = 0f
    private var ballY = 0f // X,Y = (0,0) -- origin

    //This method draws in the View
    override fun onDraw (canvas: Canvas?){
        super.onDraw(canvas)
        if (canvas == null) return

       //Parameters to draw
        val mPaint = Paint()
        mPaint.setARGB(255,128,0,128)

       canvas.drawOval(RectF(ballX, ballY, ballX+size,ballY+size), mPaint)


}
```



# Using your Custom View

You can insert your custom view into an activity's layout XML
Below we insert the a View object **BallView** into the xml of the Main Activity:

```
<!-- res/layout/activity_main.xml -->
<android.support.constraint.ConstraintLayout
    tools:context=".MainActivity">

     <edu.uw.pmpee523.a2dgraphics_bounceball.BallView
        android:id="@+id/layout_ball"
        android:layout_height="0dp"
        android:layout_width="0dp“/>

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

# Paint

Many methods accept a **Paint object**, a color to use for drawing.

- Create a Paint by specifying an alpha (opacity) value, and red/green/blue (RGB) integer values, from 0 (none) to 255 (full).

```
val panty = Paint();
panty.setARGB(alpha, red, green, blue);
```

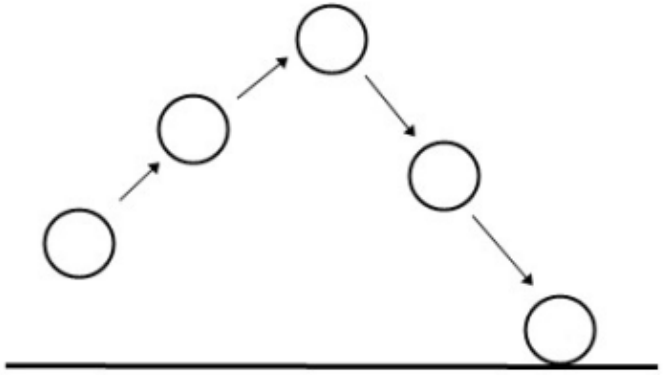
```
// example
val purple = Paint();
purple.setARGB(255, 255, 0, 255);
purple.setStyle(Style.FILL_AND_STROKE); // FILL, STROKE
```

– Paint has other useful methods like:
getTextBounds, measureText, setAlpha, setAntiAlias, setStrokeWidth, setStyle, setTextAlign, setTextSize, setTypeface

# Animation via Re-drawing

To animate a view, you must **redraw it** at regular intervals.

- On each redraw, change variables/positions of shapes.
- Force a view to redraw itself by calling its `invalidate()` method.
  - But you can't just do this in a loop; this will lock up the app's UI and lead to poor performance.



# invalidate() or postInvalidate() ?

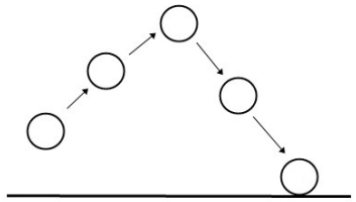
- If you want to re draw your view from **UI Thread you can call invalidate()** method.
- If you want to re draw your view from **Non UI Thread you can call postInvalidate()** method.

- If invalidate gets called it tells the system that the current view has changed and it should be redrawn as soon as possible. As this method can only be called from your UIThread another method is needed for when you are not in the UIThread and still want to notify the system that your View has been changed.

- The *postInvalidate* method notifies the system from a non-UIThread and the View gets redrawn in the next eventloop on the UIThread as soon as possible

There are some problems that arise when using postInvalidate from other threads (like not having the UI updated right-away), this will be more efficient:

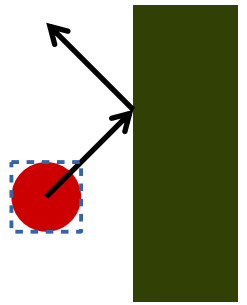
```
runOnUiThread(new Runnable() {
    public void run() {
        myImageView.setImageBitmap(image);
         imageView.invalidate();
     }
});
```



# Collision Detection

**Collision detection**:
Determining whether objects are touching each other (and reacting accordingly).

- You can calculate whether two objects have collided by seeing whether their bounds overlap.
- Android's RectF (link) and other shapes have methods to check whether they touch:
  - ***rect1***.contains(***x***, ***y***)
  - ***rect1***.contains(***rect2***)
  - RectF.intersects(***rect1***, ***rect2***)



# Google Mobile Vision API

EE P 523, Lecture 4

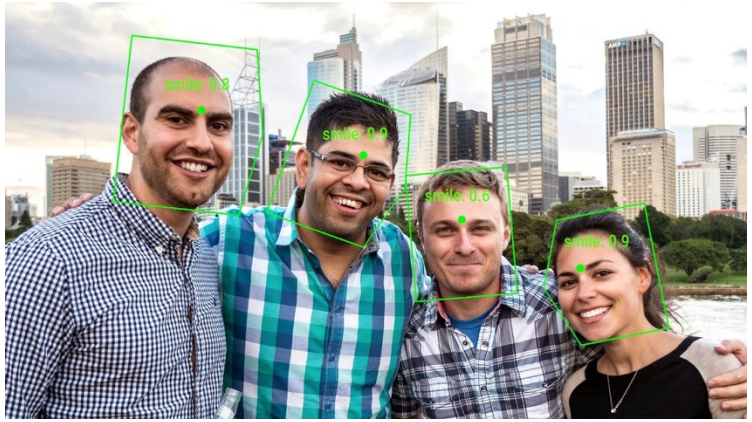
# Mobile *Application Program Interface* (API)

- It's a technical development environment that enables access to another party's application or platform.

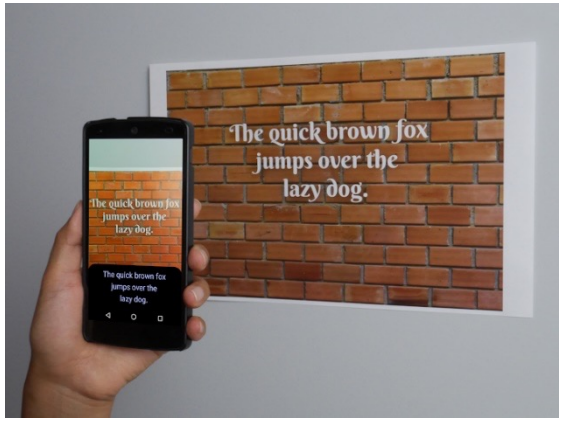
- The most famous, and most often used by mobile developers, is Facebook's API.
  Allows mobile developers limited access to the profile identity of Facebook members to verify identification on login.
  It enables Facebook members to sign up for a third-party app like Rdio or Candy Crush using their Facebook account.

- Other examples:
  - YouTube: provides access to their rich repository of video content
  - AccuWeather: offers access to weather information,
  - Flickr: enables access to a large library of photography

# Google Mobile Vision API








Note: This API is part of ML-kit (link)

# Google Vision API

The face API detects faces at a range of different angles, as illustrated below:

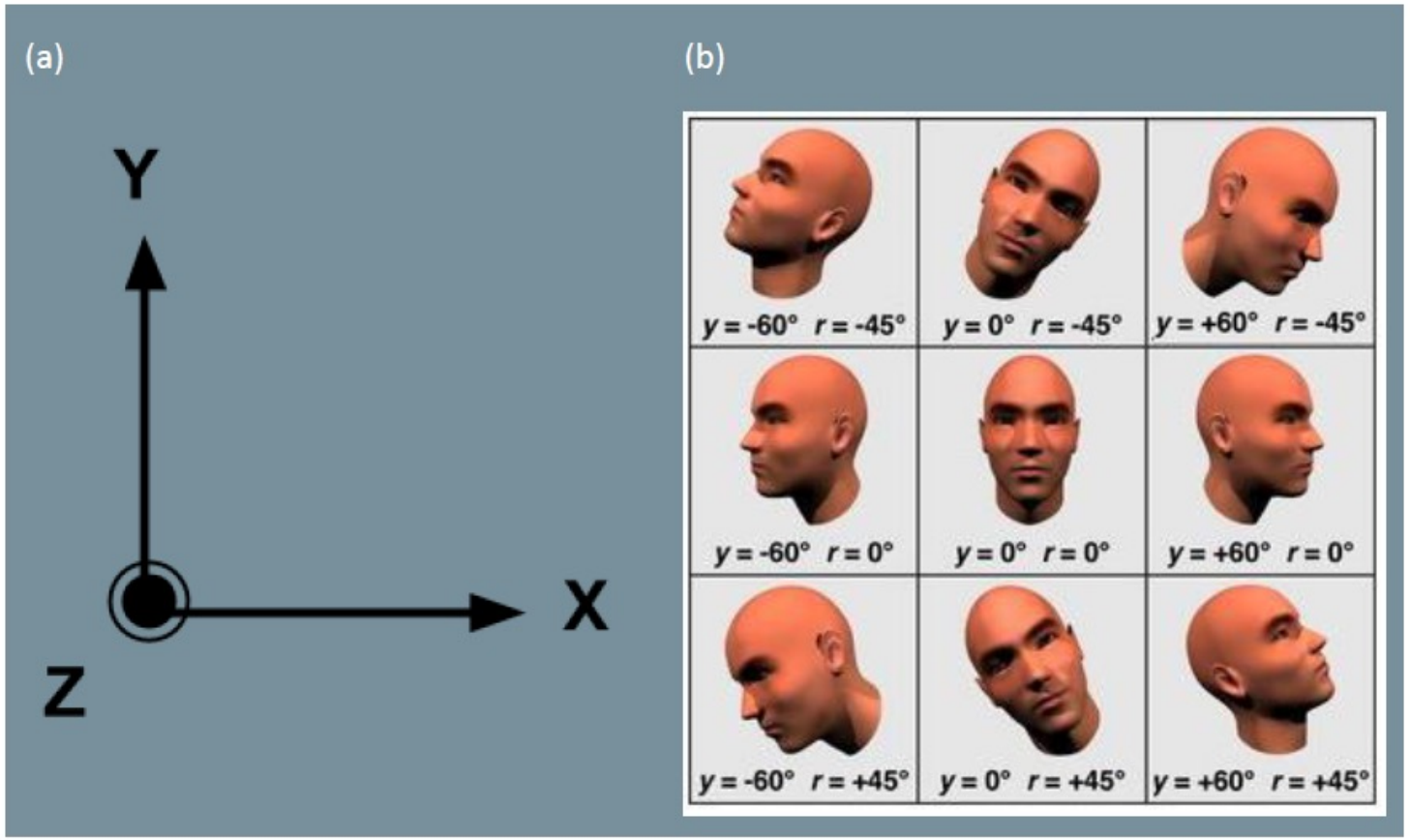


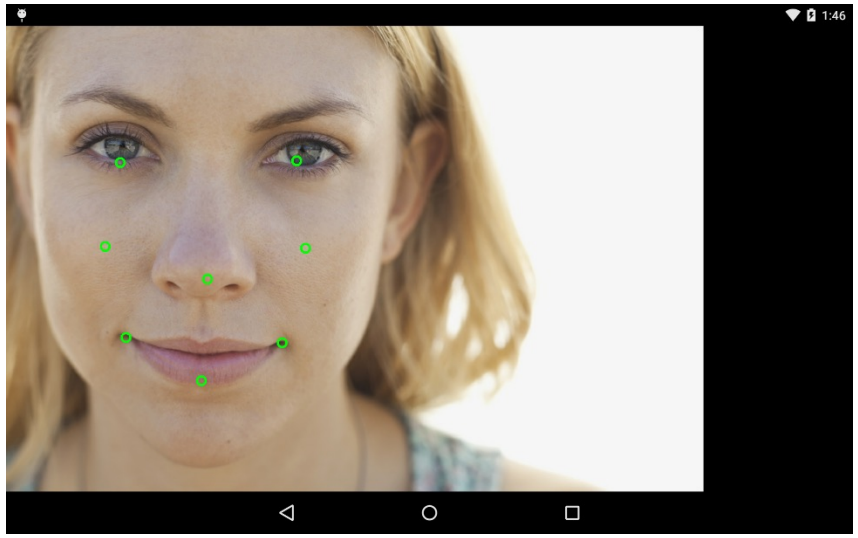

Fig. 1. Pose angle estimation. (a) The coordinate system with the image in the XY plane and the Z axis coming out of the figure. (b) Pose angle examples where y==Euler Y, r==Euler Z.

# Face Detection

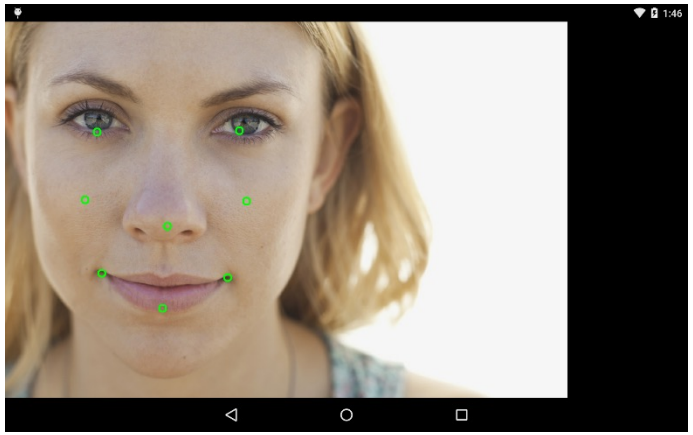
- Face Detected:
  position with an associated size and orientation.

- Once a face is detected, it can be searched for landmarks such as the eyes and nose.

- A **landmark** is a point of interest within a face. The left eye, right eye, and nose base are all examples of landmarks.
  - ✓ The Face API provides the ability to find landmarks on a detected face.



# Face Detection



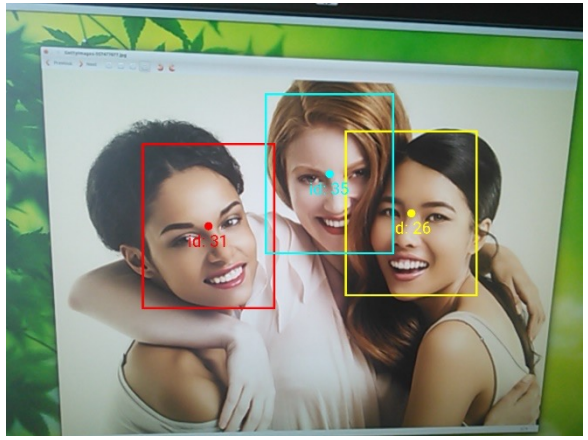
1. Create a face detector.
2. Detect faces with facial landmarks.
3. Query the face detector operational status.
4. Release the face detector.

Some methods in Face class

```
val ssp = getIsSmilingProbability ()
```

```
val reo = getIsRightEyeOpenProbability ()
```

# Face Tracking (link)

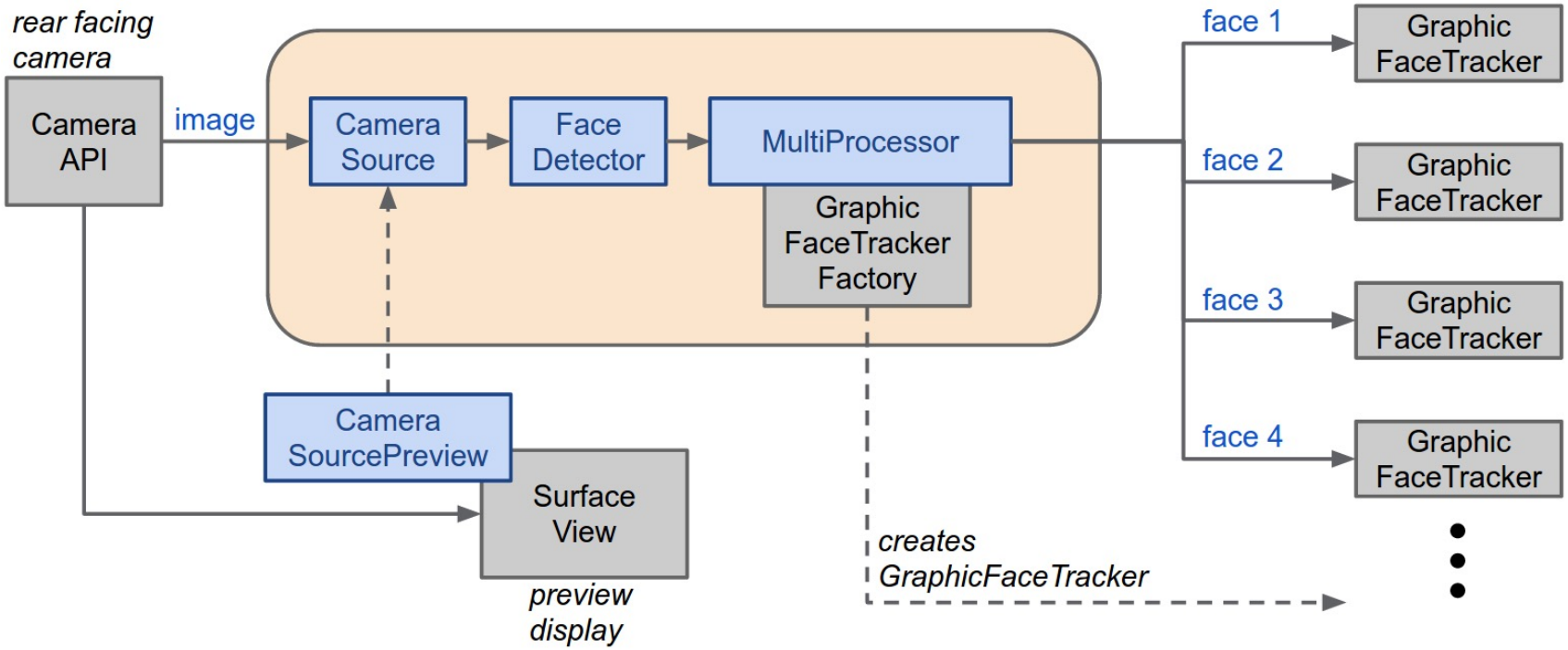



- **mode** = fast: use optimizations that favor speed over accuracy.
  For example, it may skip faces that aren't facing the camera.
- **landmarks** = none: makes face tracking faster.
- **prominent face only** = false:
  If a single face, setting this option would make face tracking faster.

**classifications** = none: Smile and eyes open.

- **tracking** = true: tracking is used to maintain a consistent ID for each face.

# Face Tracking




- Once started, the rear facing camera will continuously send preview images into the pipeline.
- The CameraSource component manages receiving these images and passing the images into the Detector, at a maximum rate of 30 frames/second

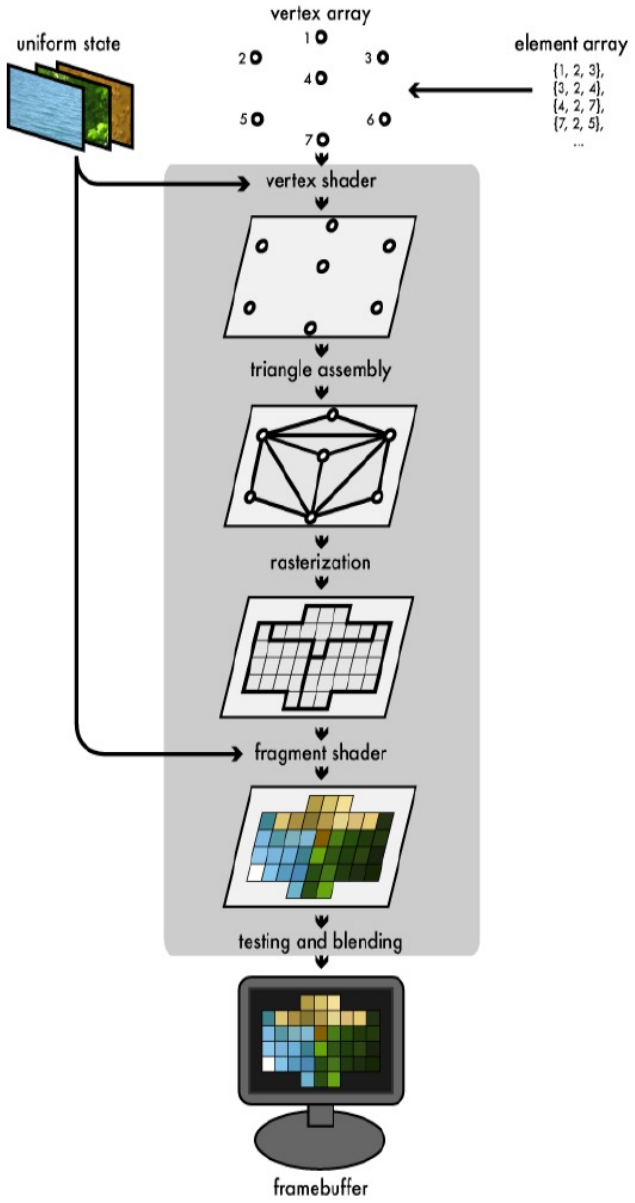
# Open GL

- High performance 2D and 3D graphics
- A cross-platform graphics API that specifies a standard software interface for 3D graphics processing hardware
- Commonly used for 3D graphics.
  - hardware-accelerated  (uses GPU, not CPU)
  - widely supported, freely available
- Android devices include a version of OpenGL for embedded systems ("ES").
- Android reference: (link)

# Open GL: 3D Graphics

**Graphics Pipeline Implemented by OpenGL**

- Host program fills OpenGL-managed memory buffers with arrays of vertices
- These vertices are projected into screen space
- Projected vertices are assembled into triangles
- Triangles are rasterized into pixels/fragments
- The fragments are assigned color values and drawn to the framebuffer



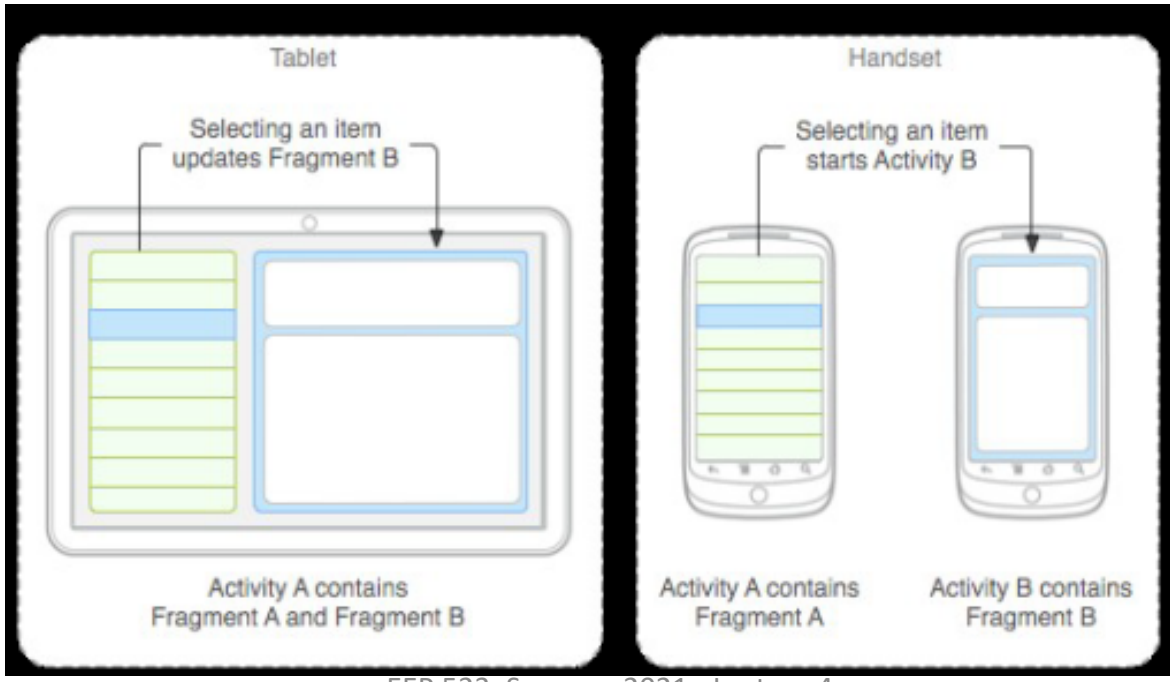
# Fragments

EE P 523, Lecture 4

# Situational Layouts

Your app can use different layout in different situations:

- different device type (tablet vs phone vs watch)
- different screen size
- different orientation (portrait vs. landscape), different country or locale (language, etc.)



# Checking Orientation in Kotlin

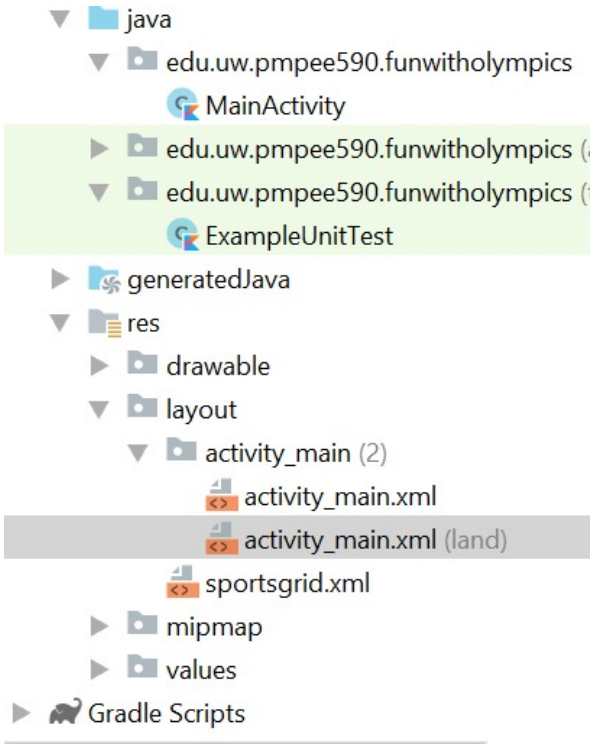
Sometimes you want your activity to behave differently in each orientation

```
//In your MainActivity.kt


onCreate(){

      if (this.resources.configuration.orientation ==
      Configuration.ORIENTATION_PORTRAIT) {
       // code for Portrait mode

      } else {
          // code for landscape mode

          }
}
```

# Situation-specific Folders

Your app will look for resource folder names with suffixes:

- screen density (e.g. **drawable-hdpi**)
  - xhdpi: 2.0 (twice as many pixels/dots per inch)
  - hdpi: 1.5
  - mdpi: 1.0 (baseline)
  - ldpi: 0.75
- screen size (e.g. layout-large) (link)
  - small, normal, large, xlarge
- orientation (e.g. layout-land)
  - portrait (), land (landscape)



# Redundant Layouts

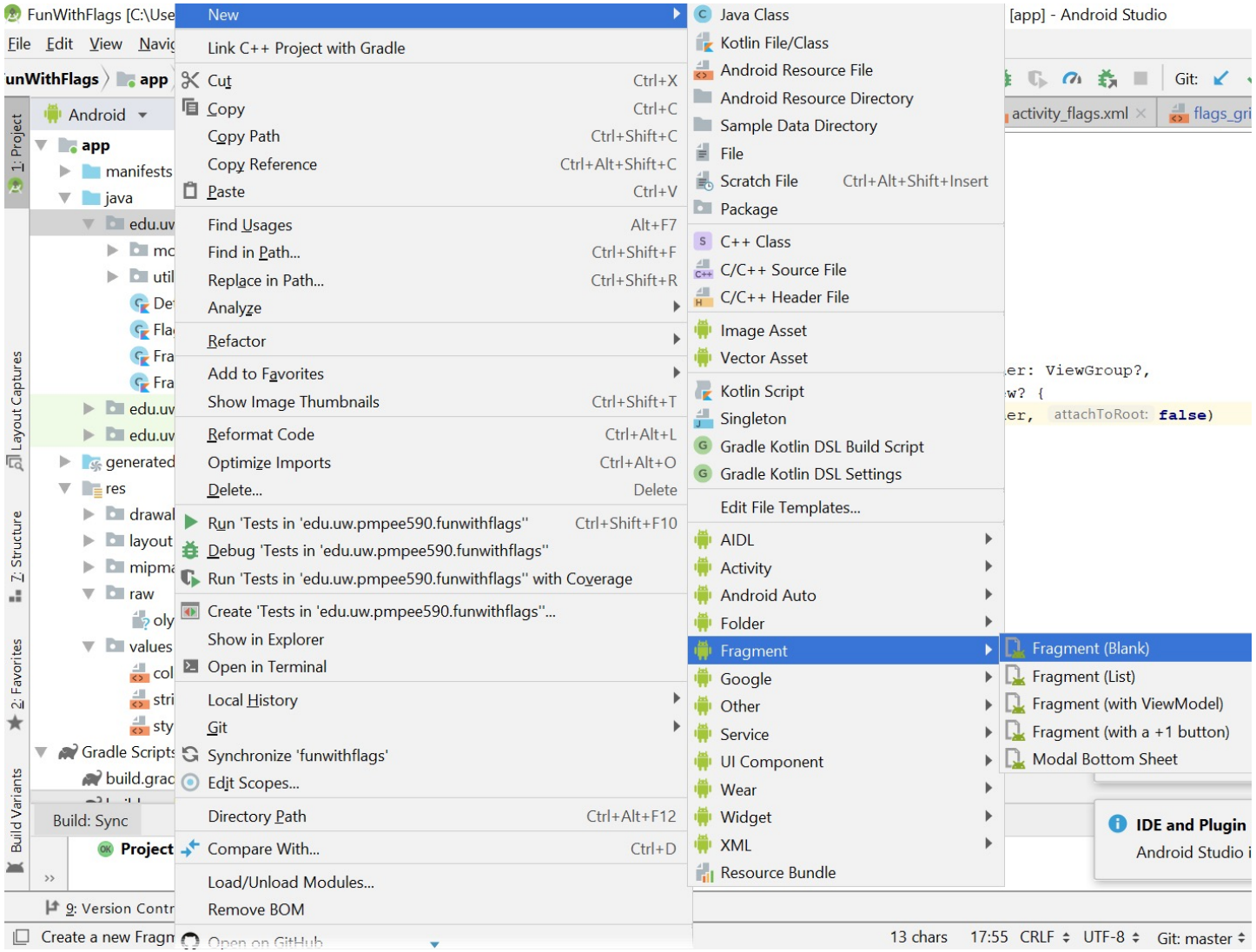
- With situational layout you begin to encounter **redundancy**.
    - The layout in one case (e.g., portrait or medium) is very similar to the layout in another case (e.g., landscape or large).
    - You don't want to represent the same XML or Java code multiple times in multiple places.
- You sometimes want your code to behave **situationally**.
    - In portrait mode, clicking a button should launch a new activity.
    - In landscape mode, clicking a button should launch a new view.

# What is a Fragment?

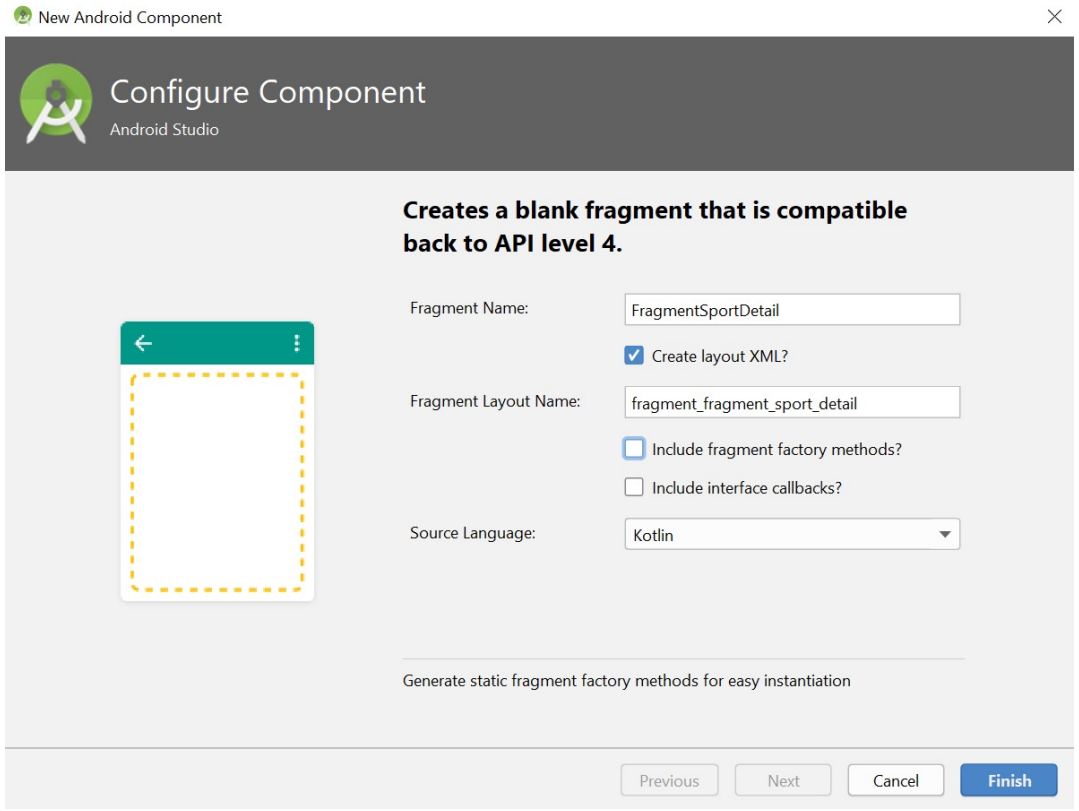
**fragment**:
A reusable segment of Android UI that can appear in an activity.

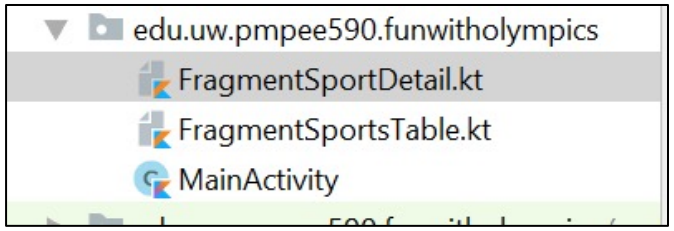
- can help handle different devices and screen sizes
- can reuse a common fragment across multiple activities
- first added in Android 3.0 *(usable in older versions if necessary)*

# How to Create a Fragment ?



# How to Create a Fragment ?




# Using Fragments in Activity XML

**Activity layout** XML can include fragments.

```
<!-- activity_main.xml -->
<LinearLayout ...>

    <fragment ...
    android:id="@+id/id1"
    android:name="ClassName1"
    tools:layout="@layout/name1" />

    <fragment ...
    android:id="@+id/id2"
    android:name="ClassName2"
    tools:layout="@layout/name2" />

</LinearLayout>
```

# Fragment Life Cycle

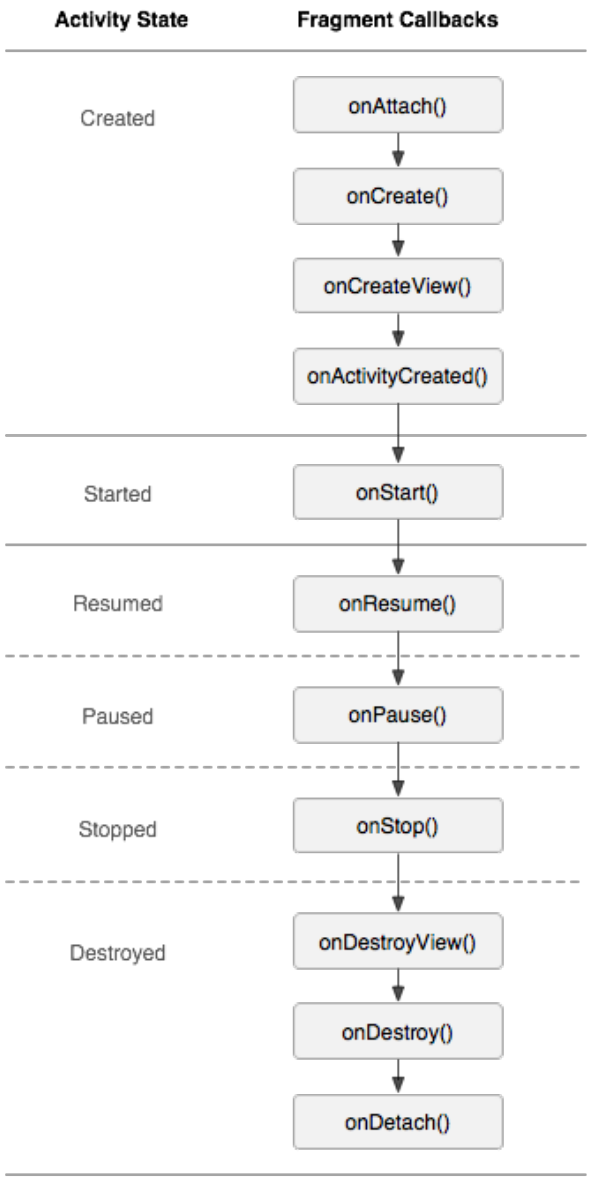
Like an activity, a fragment can exist in three states:

***Resumed***
The fragment is visible in the running activity.

***Paused***
Another activity is in the foreground and has focus, but the activity in which this fragment lives is still visible

***Stopped***
The fragment isn't visible. Either the host activity has been stopped or the fragment has been removed from the activity but added to the back stack. A stopped fragment is still alive (all state and member information is retained by the system).

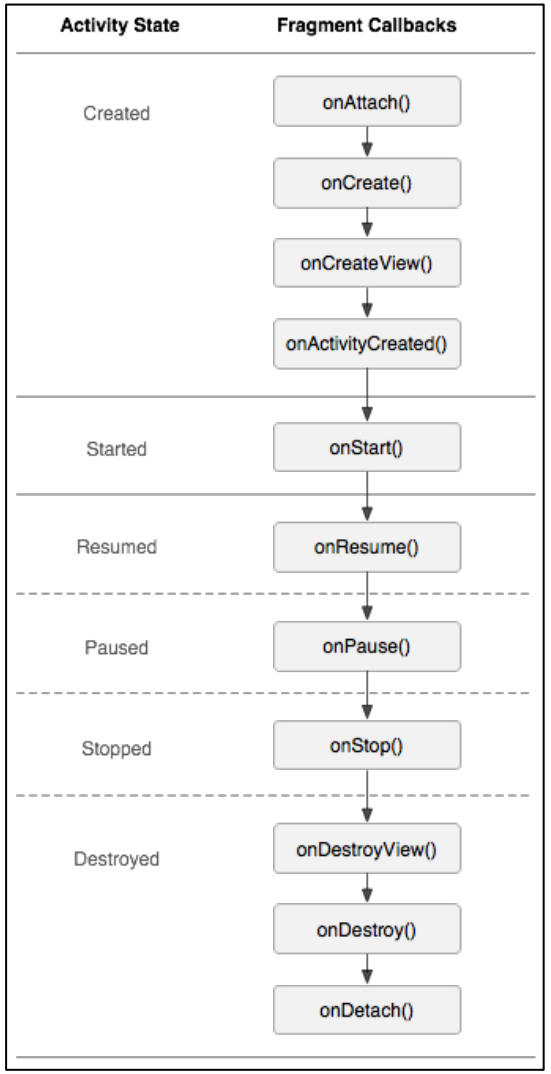
Also like an activity, you can preserve the UI state of a fragment across configuration changes and process death using a combination of onSaveInstanceState(Bundle), ViewModel, and persistent local storage. To learn more about preserving UI state, see Saving UI States.



# Fragment Life Cycle

Fragments have a similar **life cycle** and events as activities.

Important methods:
– **onAttach** to glue fragment to its surrounding activity
– **onCreate** when fragment is loading
– **onCreateView** method that must return fragment's root UI view
– **onActivityCreated** method that indicates the enclosing activity is ready
– **onPause** when fragment is being left/exited
– **onDetach** just as fragment is being deleted



# Fragment vs. Activity

Fragment code is similar to activity code, with a few changes:

– Many activity methods aren't present in the fragment, but you can call **activity** to access the activity the fragment is inside of.

```
val butt = findViewById<Button>(R.id.but);
val butt = activity!!. findViewById<Button>(R.id.but);
```

– Event handlers cannot be attached in the XML any more. :-(
Must be attached in Kotlin code instead (onClick listener).

– Passing information to a fragment (via Intents/Bundles) is trickier.
- The fragment must ask its enclosing activity for the information.
- Fragment initialization code must be mindful of order of execution.

# Fragment vs. Activity

**Activity**

```
<Button android:id="@+id/b1"
android:onClick="onClickB1" ... />
```

**xml file**

**Fragment:**

```
<Button android:id="@+id/b1" ... />
```

**xml file**

```
val button = row.getChildAt(j) as ImageButton
button.setOnClickListener(object : View.OnClickListener {
    override fun onClick(v: View?) {
```

**Fragment kotlin Class**

# Communication between Fragments

**One activity might contain multiple fragments.**

- The fragments may want to talk to each other.
  - – Use activity's getFragmentManager method.
  - – its ***findFragmentById*** method can access any fragment that has an id.

```
val orientation = activity!!.resources.configuration.orientation
if (orientation == Configuration.ORIENTATION_LANDSCAPE) {
    // code for landscape mode
    val frag =fragmentManager!!.findFragmentById(R.id.fragmentRight) as FragmentSportDetail
    frag.updateSportDetail(mSportIndex) //this mehod is in a different fragment

} else {
    //Start SportDetail activity
    startFragmentRight(mSportIndex)
}
```

# Communication between Activities and Fragments

A FragmentB (in ActivityB) can receive an intent sent by a FragmentA (in Activity)

**Kotlin file of FragmentB *(FragmentB.kt)***

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?): View? {
    // Inflate the layout for this fragment
    return inflater.inflate(R.layout.fragment_fragment_sports_table, container, false)
}
```

```
override fun onActivityCreated(savedInstanceState: Bundle?) {
    super.onActivityCreated(savedInstanceState)

    //Receive the Intent
    val intent = activity!!.intent
    val id = intent.getIntExtra(KEY, 0)
    updateSportDetail(id)
}
```

# Android Sensors

EE P 523, Lecture 4


Li-ion

# Sensors in Android

Motion sensors

Position sensors

Environment sensors

# Sensors in Android

The Android platform supports three broad categories of sensors:

**Motion sensors**

- Measure: acceleration forces and rotational forces along three axes.
- Types: accelerometers, gravity sensors, gyroscopes, and rotational vector sensors.

**Environmental sensors**

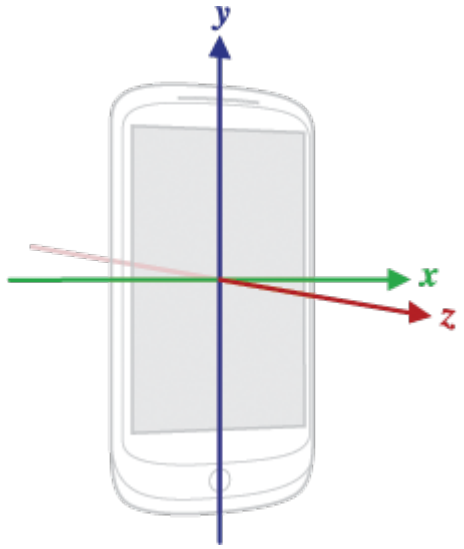
- Measure: ambient air temperature and pressure, illumination, and humidity.
- Types: barometers, photometers, and thermometers.

**Position sensors**

- Measure: physical position of a device.
- Types:  orientation sensors and magnetometers.

# Coordinate System

Coordinate system
(relative to a mobile device)
used by the Sensor API.




- The axes are not swapped when the device's screen orientation changes—**the sensor's coordinate system never changes as the device moves**

- your application must not assume that a device's natural (default) orientation is portrait. The natural orientation for many tablet devices is landscape. And **the sensor coordinate system is always based on the natural orientation of a device**.

# Accelerometer

EE P 523, Lecture 4

# Accelerometer

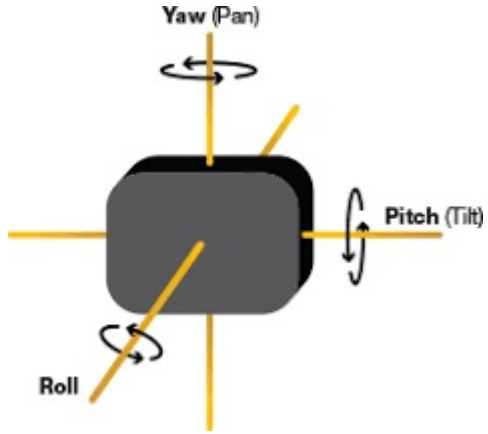
- Acceleration of the device along the 3 sensor axes.
- The measured acceleration includes both the physical acceleration (change of velocity) and the gravity.
- The measurement is reported in the x, y and z fields of `sensors_event_t.acceleration`.
- All values are in SI units **(m/s^2)** and measure the acceleration of the device minus the force of gravity along the 3 sensor axes.

# Accelerometer

**Examples:**

- The norm of (x, y, z) should be close to 0 when in free fall.
- When the device lies flat on a table and is pushed on its left side toward the right, the x acceleration value is positive.
- When the device lies flat on a table, the acceleration value along z is +9.81 also, which corresponds to the acceleration of the device (0 m/s^2) minus the force of gravity (-9.81 m/s^2).
- When the device lies flat on a table and is pushed toward the sky, the acceleration value is greater than +9.81, which corresponds to the acceleration of the device (+A m/s^2) minus the force of gravity (-9.81 m/s^2).

# Accelerometer: Typical Applications




- **Pan / Tilt / Roll (camera)**
- **Vibration / "Rough-road" detection**
  - Isolating vibration of mechanical system from outside sources
- **Vehicle skid detection**
  - Deploying "smart" braking to regain control of vehicle
- **Impact detection**
  - Detecting and logging impact, when it occurs (avoid hard drive damage in laptops)
- **Input / feedback for active suspension control systems**
  - Keeps vehicle level

# Accelerometer: Phone Applications

- **Orientation sensing**
  - landscape or portrait views of the device's screen
  - tagging the orientation to photos taken with the built-in camera

- Human activities – walking, running, dancing or skipping

- Anti-blur capturing for built-in still cameras – holding off snapping the "shutter" when camera is moving

- Image stabilization for built-in camcorders

# Accelerometer: Phone Shake

Detect shake of a phone?

1. What is shaking?
   - Fairly rapid movement of the device in one direction followed by further one in another direction, mostly but not necessarily the opposite

2. How can we detect a shake effect?
   1. register event with accelerometer: compare the current acceleration values on all three axes to the previous ones
   2. Define a threshold
   3. Detect shake:  if the values have repeatedly changed on at least two axis and those changes exceed a high enough threshold, you can clearly determine shaking.

# Accelerometer in Android

**Sensor.TYPE_ACCELEROMETER:**

| | |
|---|---|
| SensorEvent.values[0] | Acceleration force along the x axis (including gravity). |
| SensorEvent.values[1] | Acceleration force along the y axis (including gravity). |
| SensorEvent.values[2] | Acceleration force along the z axis (including gravity). |

The force of gravity is always influencing the measured acceleration

- Device sitting on a table (and not accelerating), the accelerometer reads a magnitude of $g = 9.81\ m/s^2$
- Device in free fall and therefore rapidly accelerating toward the ground at $9.81\ m/s^2$, its accelerometer reads a magnitude of $g = 0\ m/s^2$

# Accelerometer in Android

To measure the real acceleration of the device, the contribution of the force of gravity must be removed from the accelerometer data.

- Use a low-pass filter to isolate the force of gravity
- Apply a high-pass filter to remove the force of gravity

```
override fun onSensorChanged(event: SensorEvent) {
    // alpha is calculated as t / (t + dT)
    // with t, the low-pass filter's time-constant
    // and dT, the event delivery rate

    val alpha = 0.8f

    gravity[0] = alpha * gravity[0] + (1 - alpha) * event.values[0]
    gravity[1] = alpha * gravity[1] + (1 - alpha) * event.values[1]
    gravity[2] = alpha * gravity[2] + (1 - alpha) * event.values[2]

    linear_acceleration[0] = event.values[0] - gravity[0]
    linear_acceleration[1] = event.values[1] - gravity[1]
    linear_acceleration[2] = event.values[2] - gravity[2]
}
```

**t**: rough representation of the latency that the filter adds to the sensor events,

**dt**: the sensor's event delivery rate

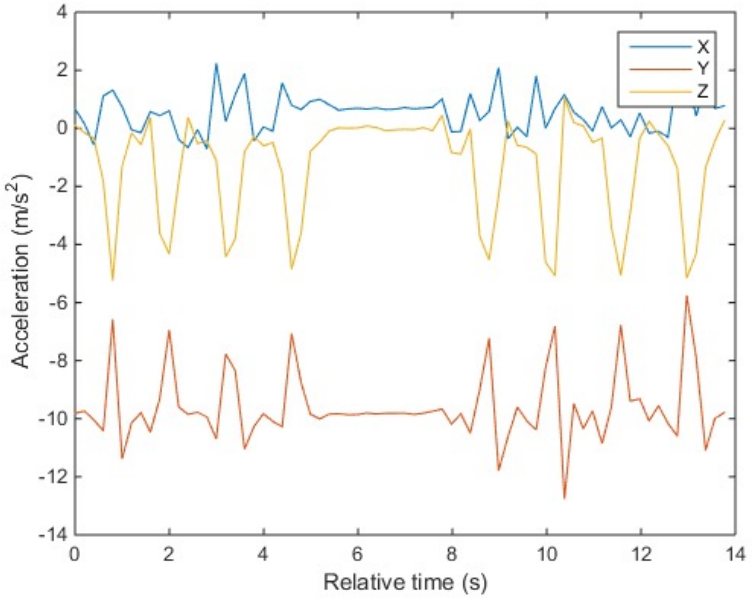
# Accelerometer in Android

- Not necessary to get accelerometer events at a very high rate
- By using a slower rate (SENSOR_DELAY_UI), we get an automatic low-pass filter, which "extracts" the gravity component of the acceleration.
- As an added benefit, we use less power and CPU resources.

```
mSensorManager.registerListener(this, mSensor, SensorManager.SENSOR_DELAY_GAME)
```

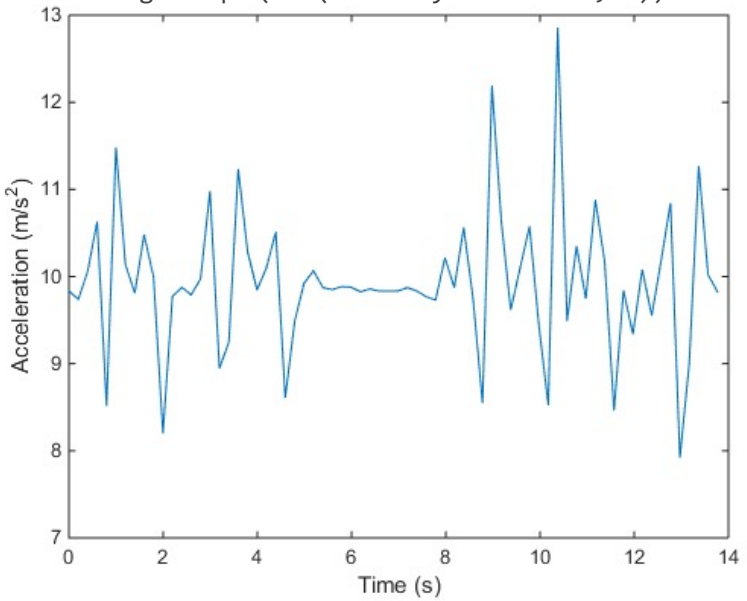
# Accelerometer: Step Counter

Raw data




Magnitude

```
mag = sqrt(sum(x.^2 + y.^2 + z.^2, 2))
```




How can we detect/count steps?

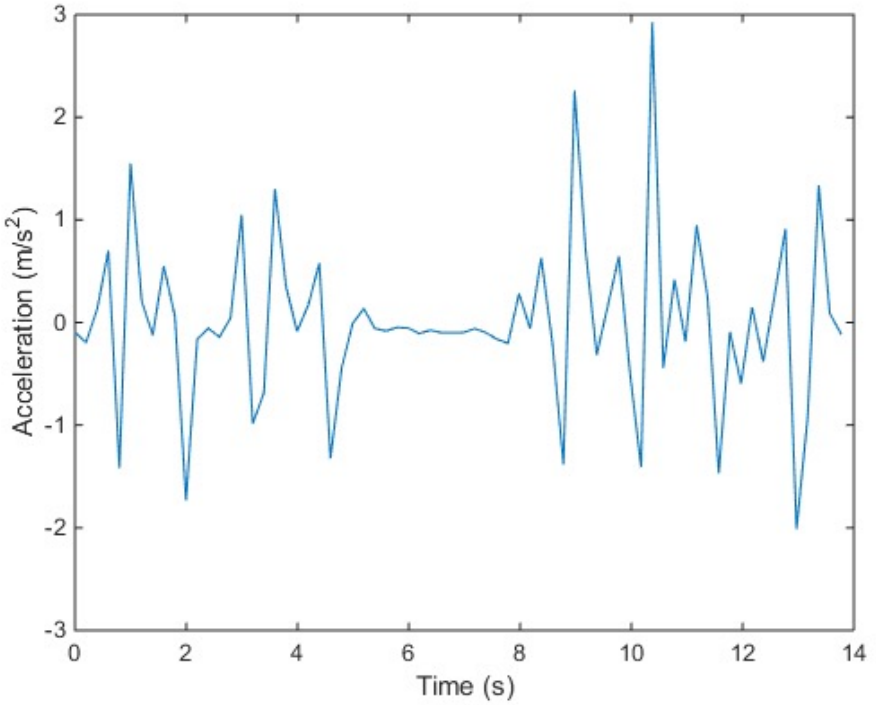
# Accelerometer: Step Counter

1. Peak finding
2. Zero crossings
3. Frequency content

# Step Counter: Peak finding

Pre-processing:

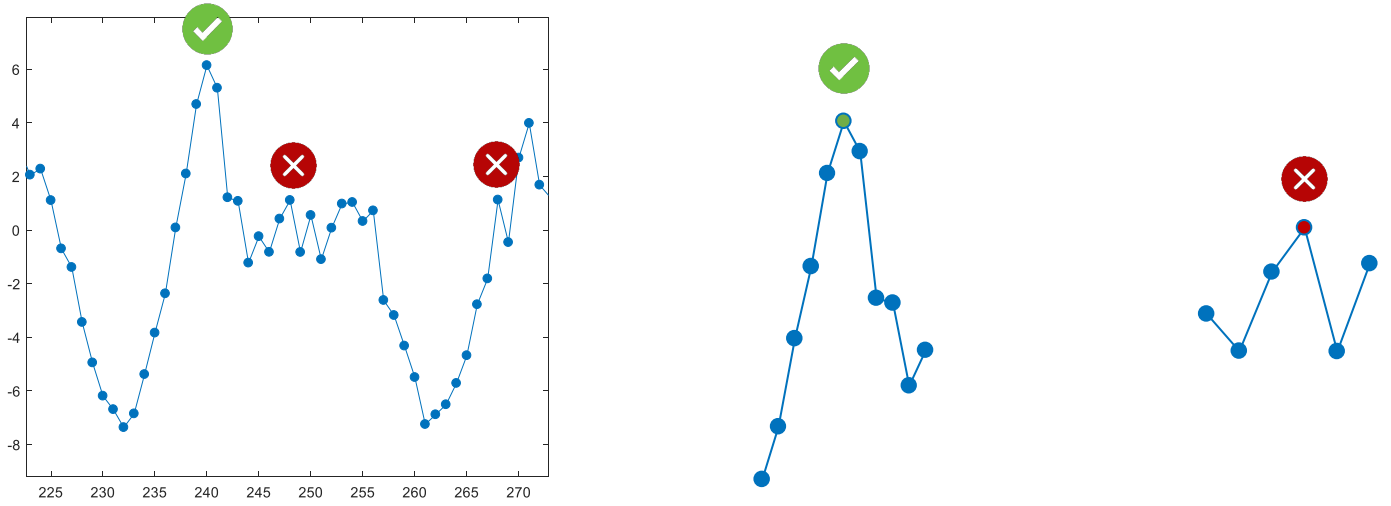
Subtracting the mean from the data will remove any constant effects, such as gravity.



# Step Counter: Peak finding

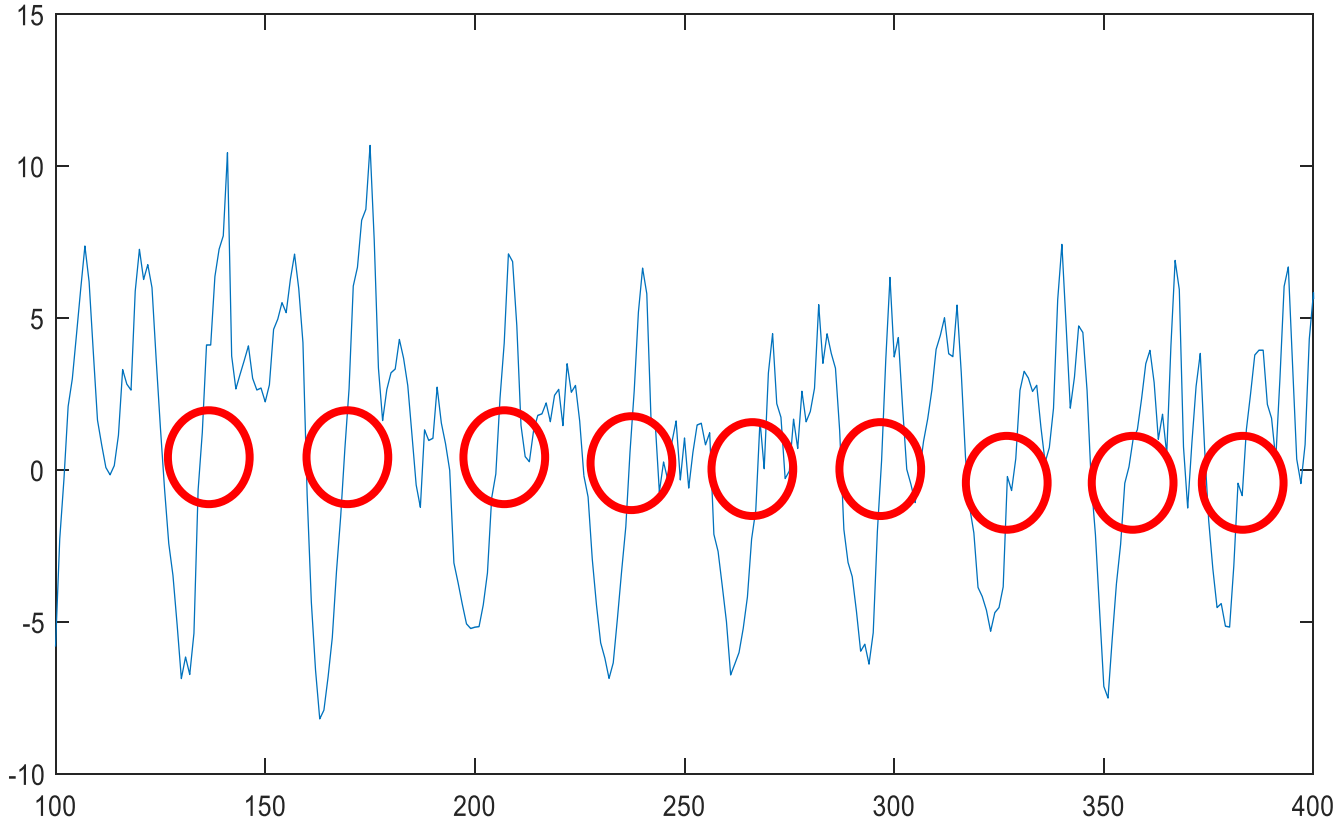
## Challenges

- Distinguishing actual peaks from local maxima
- When is a peak significant enough to indicate a step?
- How long should the window be?



# Step Counter: Zero Crossing

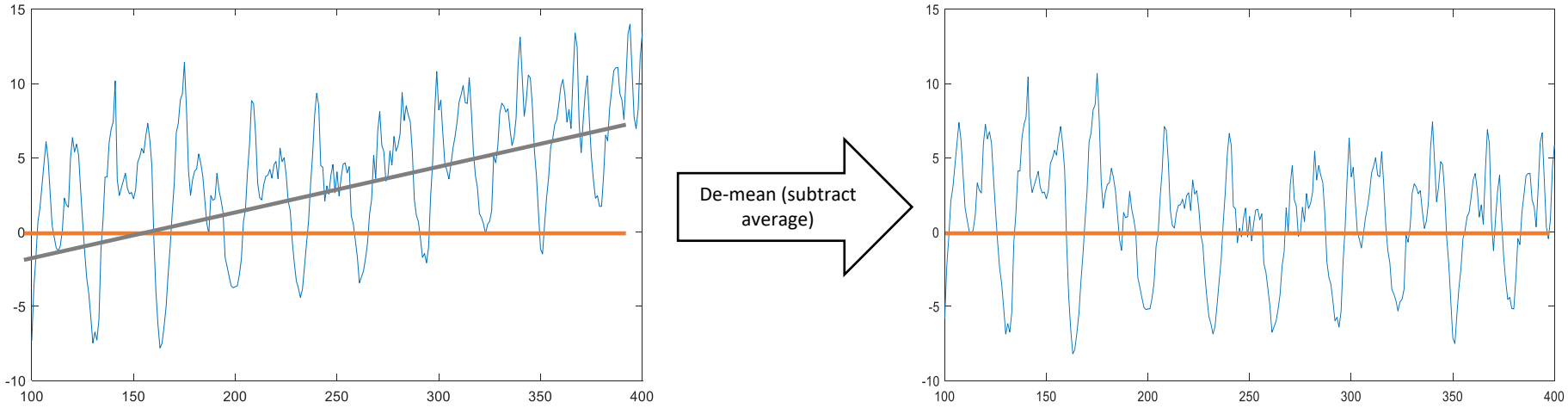
Does the signal go cross the x-axis? (note: pick increasing or decreasing)



# Step Counter: Zero Crossing

## Challenges

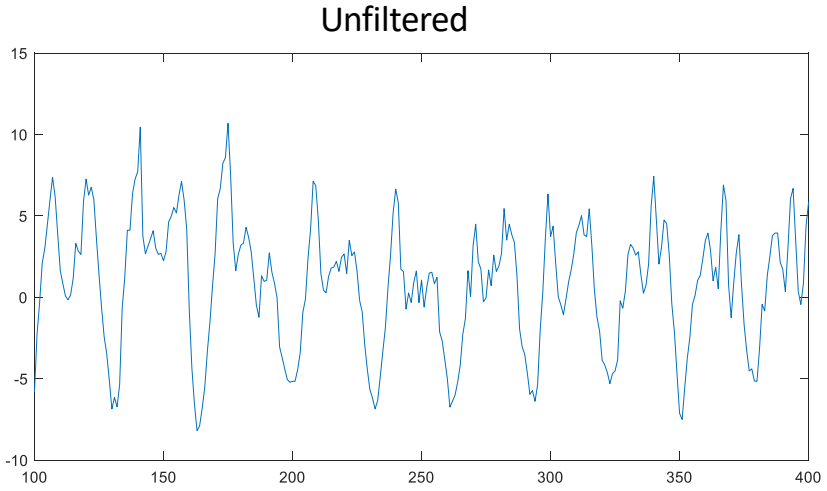
- Signal isn't always centered around zero
- False positives if there are weird perturbations in the middle

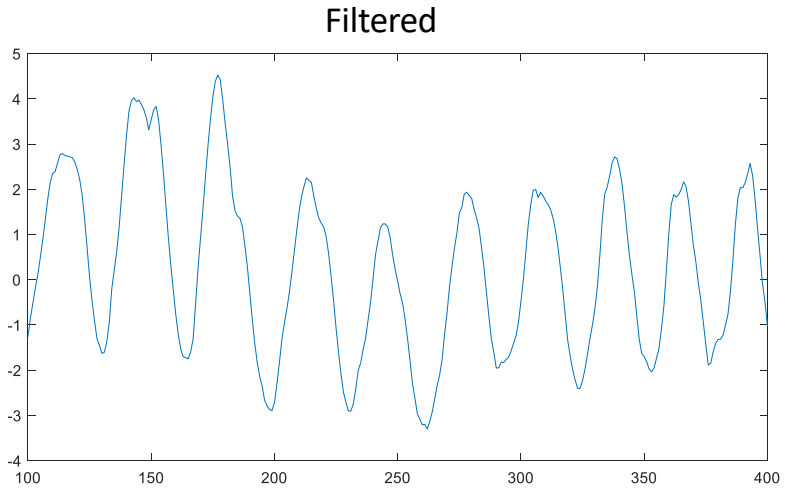


# Filters

- Signals will almost never look as clean as you want them to be

- Sensors have noise that may break assumptions that you make in your algorithm (e.g., all zero-crossings are due to steps)



# Sliding Window

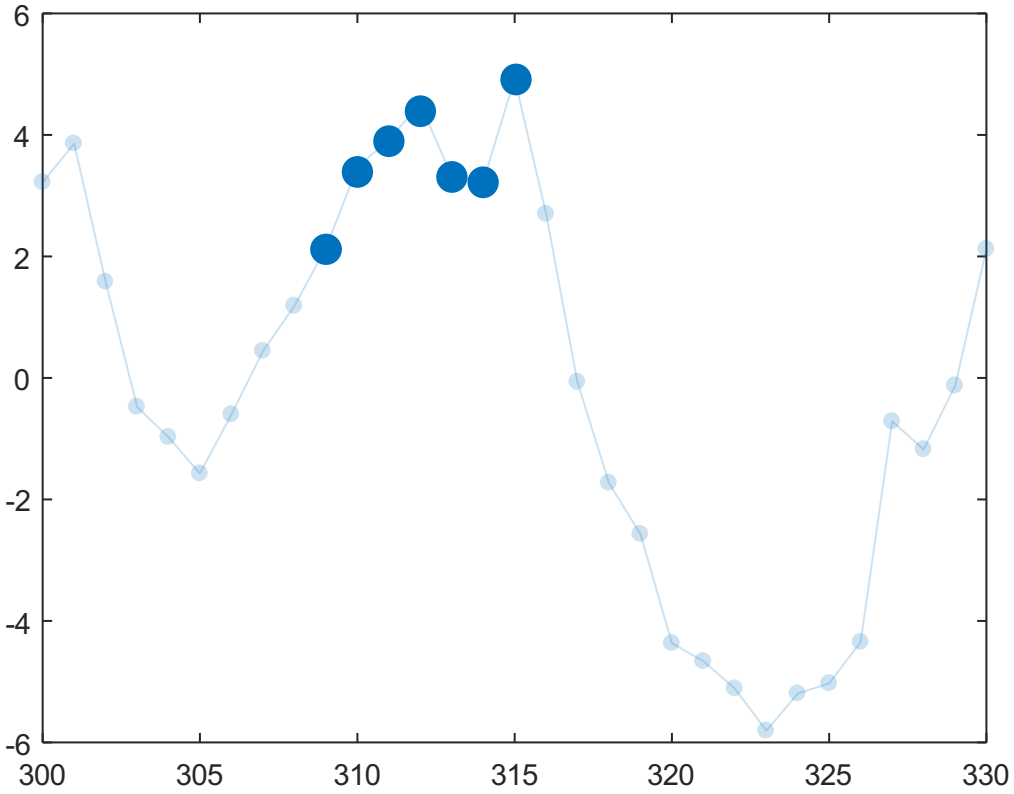
As your signal gets longer and longer, it does not make sense to analyze all of that data at once

- Infeasible for memory
- Older information is usually less useful or unneeded

You will only want to look at the most recent chunk of data

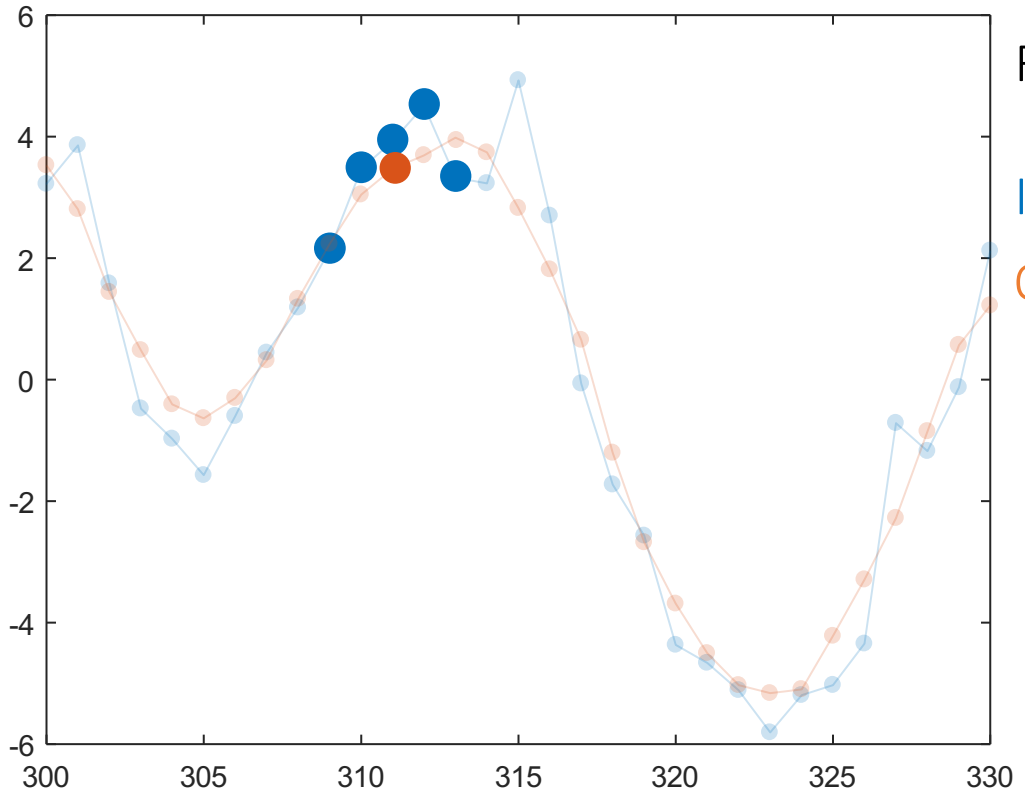
When the next value comes, add the value to your list and get rid of the oldest one

# Sliding Window




Window Size = 5

# Moving Average Filter




Filter Length [N] = 5

Input = [2.4, 4, 4.5, 5, 3.5]

Output = (2.4+4+4.5+5+3.5)/5 = 3.9

# Android Step Detector

Underlying physical sensor: **Accelerometer (+ possibly others as long as low power)**

- Reporting-mode: *Special (one event per step taken)*
- Low-power

*getDefaultSensor(SENSOR_TYPE_STEP_DETECTOR)*

- A step detector generates an event each time a step is taken by the user.
- The timestamp of the event sensors_event_t.timestamp corresponds to when the foot hit the ground, generating a high variation in acceleration.
- Compared to the step counter, the step detector should have a lower latency (less than 2 seconds).
- Both the step detector and the step counter detect when the user is walking, running and walking up the stairs.
- They should not trigger when the user is biking, driving or in other vehicles.

# Gyroscope

EE P 523, Lecture 4

# Gyroscope

Measure rotational (angular) velocity in 1, 2, or 3 directions

- 3-axis gyroscopes often implemented with a 3-axis accelerometer to provide a 6 degree-of-freedom motion tracking system

- Basic types of gyroscope:
  - Rotary (classical) gyroscopes
  - Vibrating Structure Gyroscope
  - Optical Gyroscopes

# Gyroscope

## Applications

- The gyroscope helps the accelerometer out with understanding which way your phone is orientated—it adds another level of precision so those 360-degree photo spheres really look as impressive as possible.

- Gyroscopes aren't exclusive to phones.
  They're used in altimeters inside aircraft to determine altitude and position, for example, and to keep cameras steady on the move.

# Gyroscope

## Gyroscope vs Accelerometer

- So, putting in simplest of the words –Gyroscope provides velocity, and Accelerometer provides speed.
- Gyroscope is capable to provide precision motion inside the App functionality.
- User can execute majority of the tasks with the motion of the device only.

# Gyroscope in Android

```
val sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
val sensor: Sensor? = sensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE)
```

- All values are in radians/second and measure the rate of rotation around the device's local X, Y and Z axis.

- Rotation is positive in the counter-clockwise direction. That is, an observer looking from some positive location on the x, y or z axis at a device positioned on the origin would report positive rotation if the device appeared to be rotating counter clockwise.

values[0]: Angular speed around the x-axis
values[1]: Angular speed around the y-axis
values[2]: Angular speed around the z-axis

# Gyroscope in Android

```
Override fun onSensorChanged(event: SensorEvent) {
    // This time step's delta rotation to be multiplied by the current rotation
    // after computing it from the gyro sample data.
    if (timestamp !== 0) {
        val dT = (event.timestamp - timestamp) * NS2S
        // Axis of the rotation sample, not normalized yet.
        var axisX = event.values[0]
        var axisY = event.values[1]
        var axisZ = event.values[2]

        // Calculate the angular speed of the sample
        val omegaMagnitude = sqrt((axisX * axisX + axisY * axisY + axisZ * axisZ).toDouble())

        // Normalize the rotation vector if it's big enough to get the axis
        if (omegaMagnitude > EPSILON) {
            axisX /= omegaMagnitude.toFloat()
            axisY /= omegaMagnitude.toFloat()
            axisZ /= omegaMagnitude.toFloat()
        }

        // Integrate around this axis with the angular speed by the time step
        // in order to get a delta rotation from this sample over the time step
        // We will convert this axis-angle representation of the delta rotation
        // into a quaternion before turning it into the rotation matrix.
        val thetaOverTwo = omegaMagnitude * dT / 2.0f
        val sinThetaOverTwo = sin(thetaOverTwo)
        val cosThetaOverTwo = cos(thetaOverTwo)
        deltaRotationVector[0] = (sinThetaOverTwo * axisX).toFloat()
        deltaRotationVector[1] = (sinThetaOverTwo * axisY).toFloat()
        deltaRotationVector[2] = (sinThetaOverTwo * axisZ).toFloat()
        deltaRotationVector[3] = cosThetaOverTwo.toFloat()
    }
    timestamp = event.timestamp.toInt()
    val deltaRotationMatrix = FloatArray(9)
    SensorManager.getRotationMatrixFromVector(deltaRotationMatrix, deltaRotationVector)
    // User code should concatenate the delta rotation we computed with the current
    // rotation in order to get the updated rotation.
    // rotationCurrent = rotationCurrent * deltaRotationMatrix;
}
```

Typically, the output of the gyroscope is integrated over time to calculate a rotation describing the change of angles over the time step

# Additional Base sensors

- **Compass**

- **Proximity sensor**:
  When you lift phone to your ear, the proximity sensor can turn off the display to save power and prevent accidental dialing.

- **Ambient light sensor:**
  Automatically brightens the display when you're in sunlight or a bright room and dims it in darker places.

# Composite Sensors

EE P 523, Lecture 4

# Composite Sensors

Generates data by processing and/or fusing data from one or several physical sensors -> Any sensor that is not a base sensor is called a composite sensor

Examples of composite sensors include:

- **Step detector** and **Significant motion**
  usually based on an accelerometer, but could be based on other sensors as well, if the power consumption and accuracy was acceptable.

- **Game rotation vector,** based on an accelerometer and a gyroscope.

- **Uncalibrated gyroscope**, similar to the gyroscope base sensor,
  but with the bias calibration being reported separately instead of being corrected in the measurement.

# Composite Sensors

| Sensor type | Category | Underlying physical sensors | Reporting mode |
|---|---|---|---|
| Game rotation vector | Attitude | Accelerometer, Gyroscope MUST NOT USE Magnetometer | Continuous |
| Geomagnetic rotation vector | Attitude | Accelerometer, Magnetometer, MUST NOT USE Gyroscope | Continuous |
| Glance gesture | Interaction | Undefined | One-shot |
| Gravity | Attitude | Accelerometer, Gyroscope | Continuous |
| Gyroscope uncalibrated | Uncalibrated | Gyroscope | Continuous |
| Linear acceleration | Activity | Accelerometer, Gyroscope (if present) or Magnetometer (if gyro not present) | Continuous |
| Magnetic field uncalibrated | Uncalibrated | Magnetometer | Continuous |
| Orientation (deprecated) | Attitude | Accelerometer, Magnetometer PREFERRED Gyroscope | Continuous |
| Pick up gesture | Interaction | Undefined | One-shot |
| Rotation vector | Attitude | Accelerometer, Magnetometer, AND (when present) *Gyroscope* | Continuous |
| Significant motion | Activity | Accelerometer (or another as long as very low power) | One-shot |
| Step counter | Activity | Accelerometer | On-change |
| Step detector | Activity | Accelerometer | Special |
| Tilt detector | Activity | Accelerometer | Special |
| Wake up gesture | Interaction | Undefined | One-shot |

# Reading Data from Sensors

1. Class must implement interface ***SensorEventLIstener***

```
class MainActivity : AppCompatActivity(), SensorEventListener {
```

2. Override two functions

```
override fun onAccuracyChanged(sensor: Sensor, accuracy: Int) {}
```

```
override fun onSensorChanged(event: SensorEvent) {
    if (event.sensor.type != Sensor.TYPE_ACCELEROMETER)
        return
 . . .
}
```

# Reading Data from Sensors

3. Create a ***SensorManager*** and ***Sensor*** variable

```
private lateinit var msensorManager: SensorManager
```

```
private val mAccelerometer: Sensor
```

4. Initiate an instance of the SensorManager and Sensor

```
sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
```

```
mAccelerometer = mSensorManager!!.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)
```

5. Register a listener for the particular sensor

```
mSensorManager!!.registerListener(this, mAccelerometer, SensorManager.SENSOR_DELAY_GAME)
```

# Sensor Best Practices

Be sure to unregister a sensor's listener when you are done using the sensor or when the sensor activity pauses.

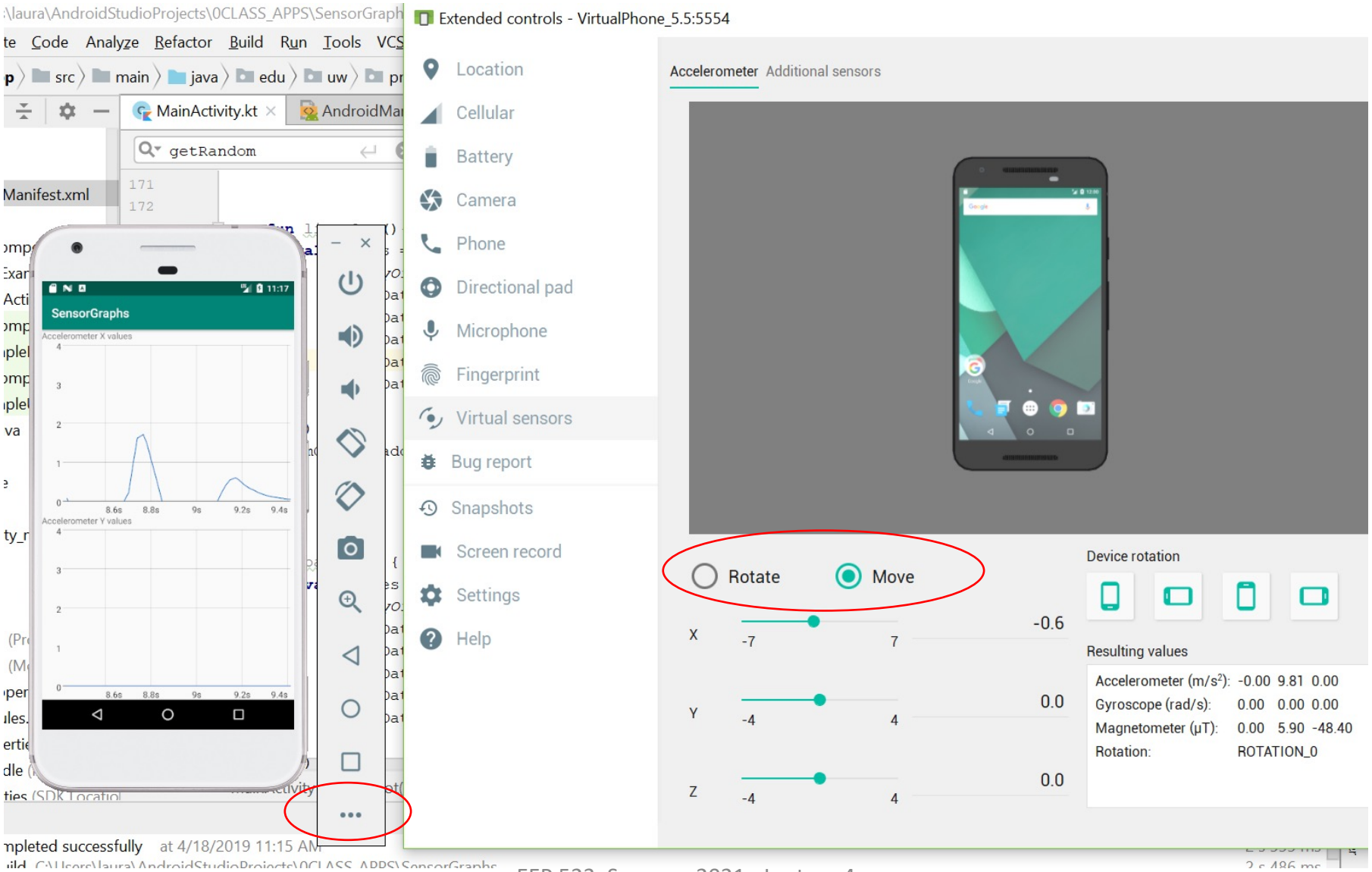
If a listener is registered and its activity is pauseD, the sensor will continue to acquire data and use the battery resources

```
override fun onPause() {
    super.onDestroy()
    mSensorManager.unregisterListener(this)
}
```

# Sensor Best Practices

- Only gather sensor data in the foreground
- On devices running Android 9 (API level 28) or higher, apps running in the background have the following restrictions:
  - Sensors that use the continuous reporting mode (link), such as accelerometers and gyroscopes, don't receive events.
  - Sensors that use the on-change or one-shot reporting modes don't receive events.
- Given these restrictions, it's best to detect sensor events either when your app is in the foreground or as part of a foreground service.

# Sensors in Emulator



# Android Supported Sensors

| Sensor | Type | Description | Common Uses |
|---|---|---|---|
| TYPE_ACCELEROMETER | Hardware | Measures the acceleration force in $m/s^2$ that is applied to a device on all three physical axes (x, y, and z), including the force of gravity. | Motion detection (shake, tilt, etc.). |
| TYPE_AMBIENT_TEMPERATURE | Hardware | Measures the ambient room temperature in degrees Celsius (°C). See note below. | Monitoring air temperatures. |
| TYPE_GRAVITY | Software or Hardware | Measures the force of gravity in $m/s^2$ that is applied to a device on all three physical axes (x, y, z). | Motion detection (shake, tilt, etc.). |
| TYPE_GYROSCOPE | Hardware | Measures a device's rate of rotation in rad/s around each of the three physical axes (x, y, and z). | Rotation detection (spin, turn, etc.). |
| TYPE_LIGHT | Hardware | Measures the ambient light level (illumination) in lx. | Controlling screen brightness. |
| TYPE_LINEAR_ACCELERATION | Software or Hardware | Measures the acceleration force in $m/s^2$ that is applied to a device on all three physical axes (x, y, and z), excluding the force of gravity. | Monitoring acceleration along a single axis. |

# Android Supported Sensors

| | | | |
|---|---|---|---|
| `TYPE_MAGNETIC_FIELD` | Hardware | Measures the ambient geomagnetic field for all three physical axes (x, y, z) in μT. | Creating a compass. |
| `TYPE_ORIENTATION` | Software | Measures degrees of rotation that a device makes around all three physical axes (x, y, z). As of API level 3 you can obtain the inclination matrix and rotation matrix for a device by using the gravity sensor and the geomagnetic field sensor in conjunction with the `getRotationMatrix()` method. | Determining device position. |
| `TYPE_PRESSURE` | Hardware | Measures the ambient air pressure in hPa or mbar. | Monitoring air pressure changes. |
| `TYPE_PROXIMITY` | Hardware | Measures the proximity of an object in cm relative to the view screen of a device. This sensor is typically used to determine whether a handset is being held up to a person's ear. | Phone position during a call. |

# Android Supported Sensors

| | | | |
|---|---|---|---|
| TYPE_RELATIVE_HUMIDITY | Hardware | Measures the relative ambient humidity in percent (%). | Monitoring dewpoint, absolute, and relative humidity. |
| TYPE_ROTATION_VECTOR | Software or Hardware | Measures the orientation of a device by providing the three elements of the device's rotation vector. | Motion detection and rotation detection. |
| TYPE_TEMPERATURE | Hardware | Measures the temperature of the device in degrees Celsius (°C). This sensor implementation varies across devices and this sensor was replaced with the TYPE_AMBIENT_TEMPERATURE sensor in API Level 14 | Monitoring temperatures. |

# Libraries in Android

EE P 523, Lecture 4

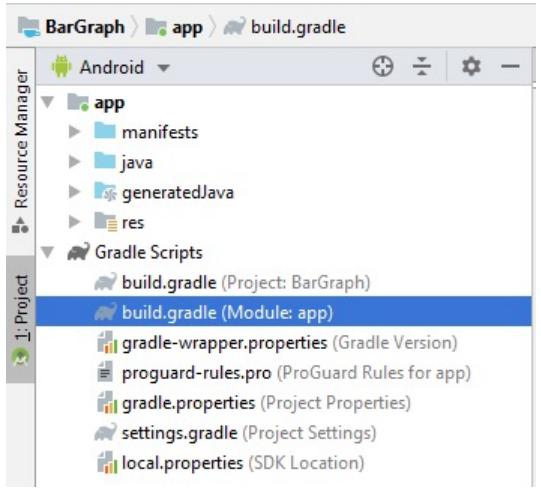
# Libraries

- Many Android developers have produced useful **libraries**.
- There is a **Maven repository** to store various libraries.
- This makes it easy to add them to your Android Studio projects.
- Most libraries use permissive licenses so that you can use them for free and can include them in the code of commercial apps/products.
- (Some libraries must be downloaded as .JARs and added manually to your project)

# Adding a library to your project

- Edit the **build.gradle** file for your 'app' module and add lines to the following section at the bottom.

- You can usually find out what file name to write below by going to various libraries' home pages / GitHub pages.




```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    implementation 'com.android.support:appcompat-v7:28.0.0'
    implementation 'com.android.support.constraint:constraint-layout:1.1.3'
    implementation 'com.android.support:support-v4:28.0.0'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.2'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'

    implementation 'com.jjoe64:graphview:4.2.2'

}
```

# Picasso Library

- **Picasso** is a powerful library for manipulating images.
  - written by Square, inc.
  - http://square.github.io/picasso/
- To add Picasso to your project:

```
// in build.gradle dependencies { ... compile 'com.squareup.picasso:picasso:2.5.2' }
```

```
<!-- in AndroidManifest.xml --> <uses-permission android:name="android.permission.INTERNET" />
```

# Picasso Library

In your app's Kotlin code, write:

```
Picasso.with(this) .load("url") .into(ImageView)
```

## Example

```
ImageView img = (ImageView) findViewById(R.id.photo);
Picasso.with(this)
      .load("http://i.imgur.com/DvpvklR.png")
      .into(img);
```

# Picasso Library

| Method | Description |
|---|---|
| `centerCrop()` | center and crop image inside view |
| `centerInside()` | resize image proportionally inside view |
| `error(id)` | show given drawable as error |
| `fetch()` | download image in the background |
| `fit()` | resize image to fit view bounds |
| `get()` | return image as a Bitmap |
| `into(view)` | puts image into given view |
| `placeholder(id)` | show given drawable while loading |
| `resize(width, height)` | change image size in pixels |
| `rotate(degrees)` | rotate clockwise |
| `tag("tag")` | attaches a "tag" to a loading image (useful for bulk operations shown later) |
| `transform(trans)` | apply complex transformations |

# Picasso Library

| Method | Description |
|---|---|
| `cancelRequest(view)` | abort any image loading in that view |
| `cancelTag("tag")` | cancel all images with given tag |
| `invalidate("url")`<br>`invalidate(File)` | flush out cache of given image,<br>so it will be re-downloaded the next time |
| `load("url")`<br>`load(id)`<br>`load(File)` | load an image from various sources |
| `pauseTag("tag")` | pause all image loads for given tag |
| `resumeTag("tag")` | unpause all image loads for given tag |
| `shutdown()` | stop entire Picasso system |
| `with(context)` | use given activity/fragment as context |

# Libraries of Interest

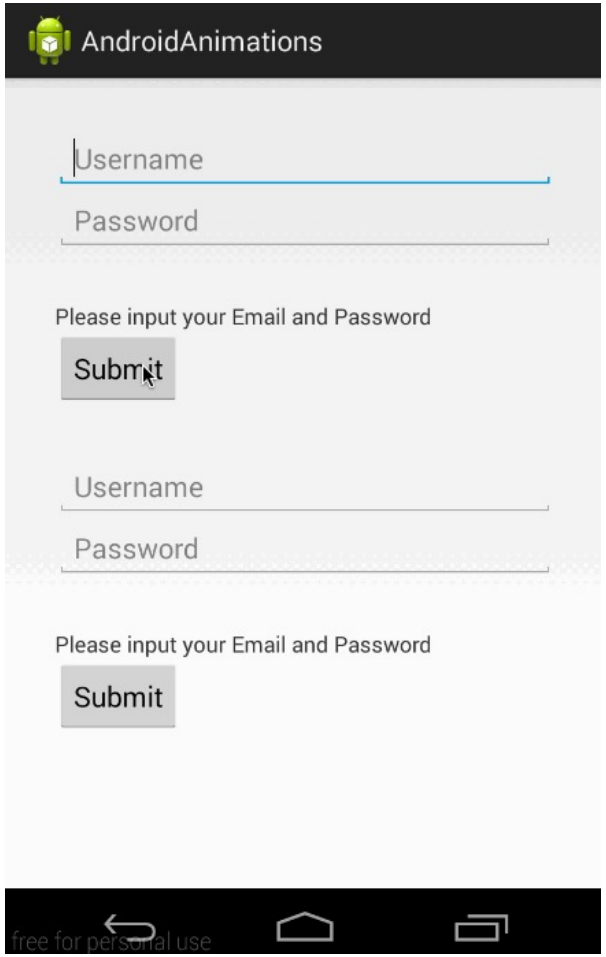
- **Android-Bootstrap**
  **C**ustomizable widgets not normally available in Android
  https://github.com/Bearded-Hen/Android-Bootstrap

- **Ion**
  make it easier to download files from the web.
  https://github.com/koush/ion

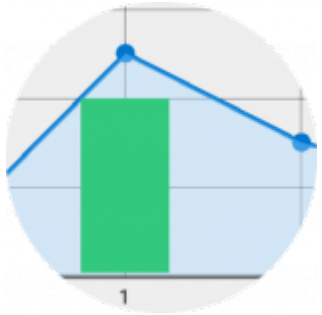
- **ButterKnife**
  Popular library intended to simplify usage of Android widgets and events in Java code.
  http://jakewharton.github.io/butterknife/

- Ambitious Android user named ***daimajia*** has created several libraries, including one to do animation effects on Views.
  https://github.com/daimajia/AndroidViewAnimations

# Animation Library



# Graph View




**Open-source graph plotting library for Android**

- Adding GraphView to your project:
  Add that line to your ***build.gradle*** file under your **app** directory into the dependencies block:

```
implementation 'com.jjoe64:graphview:4.2.2'
```

# Graph View: Line graph

```
fun lineplot(){
   val series = LineGraphSeries(
        arrayOf(
            DataPoint(0.0, 1.0),
            DataPoint(1.0, 5.0),
            DataPoint(2.0, 3.0),
            DataPoint(3.0, 2.0),
            DataPoint(4.0, 6.0)
        )
    )
    mGraphX.addSeries(series)
}
```

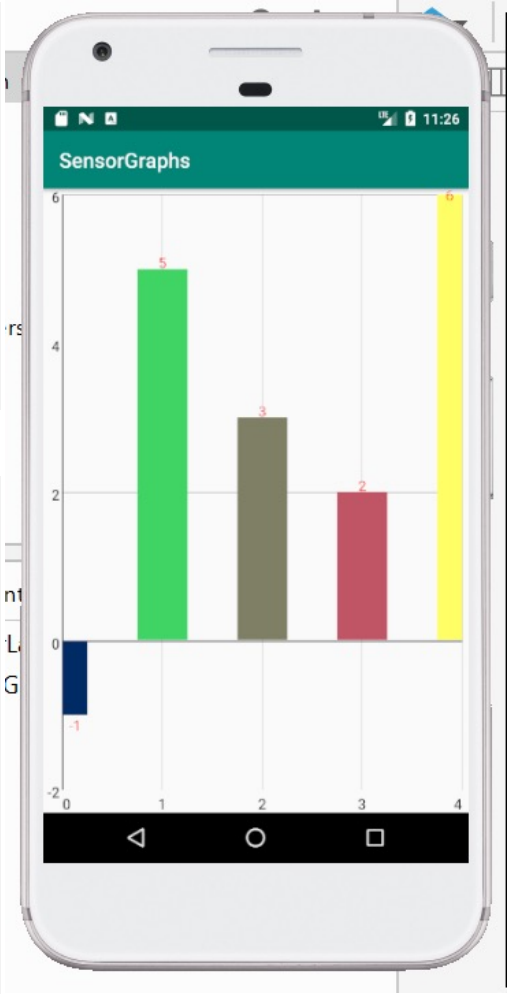
# Graph View: Bar graph

```
fun barplot(){
    val series = BarGraphSeries(
        arrayOf(
            DataPoint(0.0, -1.0),
            DataPoint(1.0, 5.0),
            DataPoint(2.0, 3.0),
            DataPoint(3.0, 2.0),
            DataPoint(4.0, 6.0)
        )
    )
    mGraphX.addSeries(series)

      // styling
    series.setValueDependentColor {data ->Color.rgb(data.x.toInt() * 255 / 4,
            Math.abs(data.y * 255 / 6).toInt(),100
        )
    }
    series.spacing = 50

   // draw values on top
    series.isDrawValuesOnTop = true
    series.valuesOnTopColor = Color.RED
}
```

# Graph View

```
fun barplot(){ //call this function anywhere, on the onCreate for example
    val series = BarGraphSeries(
        arrayOf(
            DataPoint(0.0, -1.0),
            DataPoint(1.0, 5.0),
            DataPoint(2.0, 3.0),
            DataPoint(3.0, 2.0),
            DataPoint(4.0, 6.0)     )
    )
    mGraphX.addSeries(series)
```

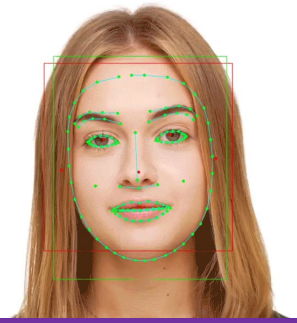
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
              android:orientation="vertical"
        tools:context=".MainActivity">
    <com.jjoe64.graphview.GraphView
            android:layout_weight="1"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:id="@+id/mGraphX"/>
</LinearLayout>
```

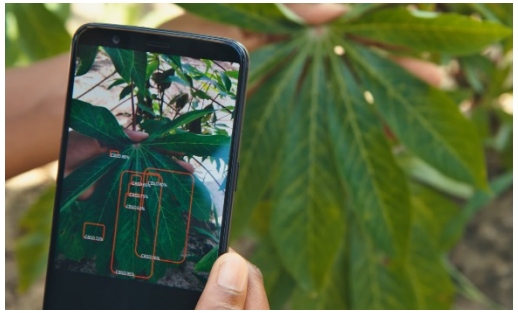


# Graph View: Real Time graph

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    mSeriesXaccel = LineGraphSeries()
    mSeriesYaccel = LineGraphSeries()
    initGraphRT(mGraphX,mSeriesXaccel!!)
    initGraphRT(mGraphY,mSeriesYaccel!!)
}
```

```
override fun onSensorChanged(event: SensorEvent) {

    if (event.sensor.type != Sensor.TYPE_ACCELEROMETER)
        return
    linear_acceleration[0] = event.values[0]
    linear_acceleration[1] = event.values[1]
    linear_acceleration[2] = event.values[2]


    val xval = System.currentTimeMillis()/1000.toDouble()//graphLastXValue += 0.1
    mSeriesXaccel!!.appendData(DataPoint(xval, linear_acceleration[0].toDouble()),
true, 50)
    mSeriesYaccel!!.appendData(DataPoint(xval, linear_acceleration[1].toDouble()),
true, 50)
}
```





# Tensor Flow Lite

**Mobile Applications**
**for Sensing and Control**

# Tensor Flow *Lite* (link)

**TensorFlow**: end-to-end open-source platform for machine learning.

**TensorFlow Lite:** lightweight solution for mobile and embedded devices.

- It enables on-device machine learning inference with low latency and a small binary size.

- TensorFlow Lite also supports hardware acceleration with the Android Neural Networks API.

# Tensor Flow *Lite* (link)

It's not *yet* designed for training models!

- You train a model on a higher powered machine,
- then convert that model to the .TFLITE format, from which it is loaded into a mobile interpreter.

# Tensor Flow *Lite* (link)

**Why do we need a new mobile-specific library?**

The next wave of machine learning applications will have significant processing on mobile and embedded devices.

- Machine Learning is changing the computing paradigm.
- Consumer expectations are also trending toward natural, human-like interactions with their devices, driven by the camera and voice interaction models.

# Tensor Flow *Lite* (link)

**Why do we need a new mobile-specific library?**

- Innovation at the *silicon layer* is enabling new possibilities for hardware acceleration, and frameworks such as the Android Neural Networks API make it easy to leverage these.

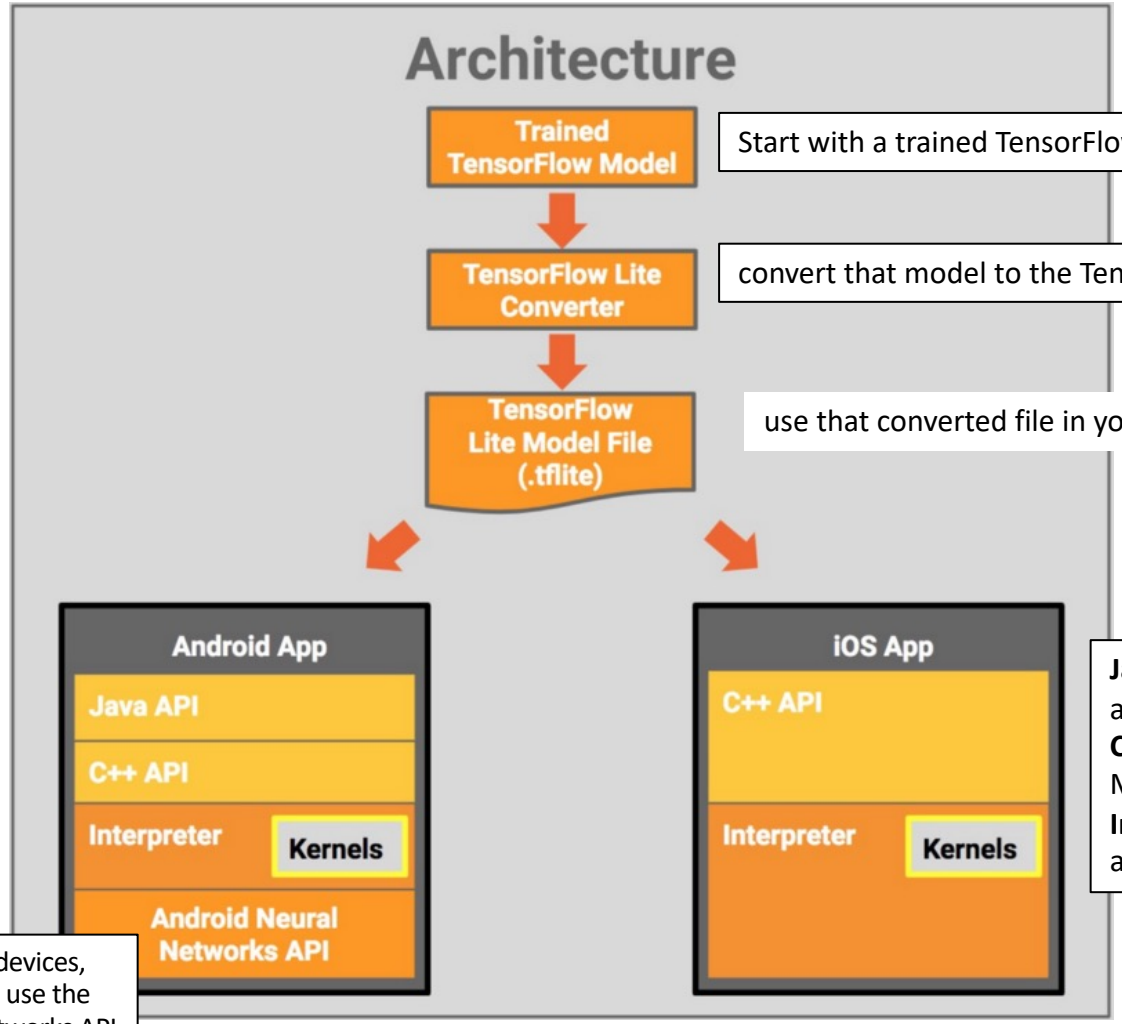
- Recent advances in real-time computer-vision and spoken language understanding have led to mobile-optimized benchmark models being open sourced (e.g. MobileNets, SqueezeNet).

- Widely-available smart appliances create new possibilities for on-device intelligence.

- Interest in stronger user data privacy paradigms where user data does not need to leave the mobile device.

- Ability to serve 'offline' use cases, where the device does not need to be connected to a network.

# Tensor Flow *Lite* (link)




Start with a trained TensorFlow model on disk

convert that model to the TensorFlow Lite file format (.tflite)

use that converted file in your mobile application

**Java API**: A convenience wrapper around the C++ API on Android.
**C++ API**: Loads the TensorFlow Lite Model File and invokes the Interpreter
**Interpreter:** Executes the model using a set of kernels

On select Android devices, the Interpreter will use the Android Neural Networks API for hardware acceleration, or default to CPU execution if none are available.

# Tensor Flow *Lite* (link)

- Create a basic model (link)

# HOMEWORK HINTS

# Assignment: Colliding Balls

**Option 1:**

Re-draw the ball according the data read from the accelerometer

**Option 2:**

Draw the ball once, and translate the coordinates of the canvas
according to the data read form the accelerometer and the screen rotation

# Assignment: Colliding Balls

**Option 2**: Draw the ball once, and translate the coordinates of the canvas according to the data read form the accelerometer and the screen rotation

In your kotlin main activity

```
public override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        //Optional: set the flag to keep the screen on
        WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON


        // Get an instance of the SensorManager: mSensorManager
        ................


        // Get an instance of the WindowManager
        mWindowManager = getSystemService(Context.WINDOW_SERVICE) as WindowManager
        mDisplay = mWindowManager!!.defaultDisplay

        // instantiate our simulation view and set it as the activity's content
        mSimulationView = SimulationView(this, mSensorManager, mWindowManager, mDisplay)

        //Add simulaton view
        // if you want to add a background color:  mSimulationView!!.setBackgroundColor(Color.rgb(r,g,b))
        setContentView(mSimulationView)
    }
```

# Assignment: Colliding Balls

In your kotlin simulation class

```
private val mDstWidth: Int
    private val mDstHeight: Int
    private val mAccelerometer: Sensor
    private val mXDpi: Float
    private val mYDpi: Float
    private val mMetersToPixelsX: Float
    private val mMetersToPixelsY: Float
    private var mXOrigin= 0.toFloat()
    private var mYOrigin= 0.toFloat()
    private var mSensorX= 0.toFloat()
    private var mSensorY= 0.toFloat()
    private var mHorizontalBound= 0.toFloat()
    private var mVerticalBound= 0.toFloat()
    private var mBallView: BallView
    private var mSensorManager: SensorManager
    private var mDisplay: Display

class SimulationView(context: Context, sensorManager: SensorManager, windowManager: WindowManager, display: Display):FrameLayout(context), SensorEventListener
{

    // diameter of the ball in meters
    val sBallDiameter = 0.004f

    init {
        mSensorManager = sensorManager
        mDisplay = display
        mAccelerometer = mSensorManager!!.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)

        val metrics = DisplayMetrics()
        windowManager.defaultDisplay.getMetrics(metrics)
        mXDpi = metrics.xdpi
        mYDpi = metrics.ydpi
        mMetersToPixelsX = ...
        mMetersToPixelsY = ...

        // get the width and heigh of the ball in meters
        mDstWidth =
        mDstHeight =
        mBallView = BallView(context, this, mDstWidth, mDstHeight)
       }
```

# Assignment: Colliding Balls

In your kotlin simulation class

```
/*    compute the origin of the screen relative to the origin of
     the bitmap*/
fun screenOriginBitmap(){
    mXOrigin = (width - mDstWidth) * 0.5f
    mYOrigin = (height - mDstHeight) * 0.5f
    mHorizontalBound = (width / mMetersToPixelsX - sBallDiameter)
    mVerticalBound = (height / mMetersToPixelsY - sBallDiameter)   }

override fun onSensorChanged(event: SensorEvent) {
    if (event.sensor.type != Sensor.TYPE_ACCELEROMETER)
        return
    /*
     * record the accelerometer data, the event's timestamp as well as the current time. The current time is needed so we can calculate the
     * "present" time during rendering. In this application, we need to take into account how the screen is rotated with respect to the
     * sensors (which always return data in a coordinate space aligned to with the screen in its native orientation).
     */

    when (mDisplay!!.rotation) {
        Surface.ROTATION_0 -> {
            mSensorX = event.values[0]
            mSensorY = event.values[1]
        }
        ...........
       //TO DO FOR THE DIFFEREENTS ROTATIN ANGLES: 90,180,270
        ..........
    }

    /*    compute the origin of the screen relative to the origin of the bitmap*/
     screenOriginBitmap()
}
```

# Assignment: Colliding Balls

In your kotlin simulation class

```
 /*   avoid doing any calculation in the onDraw method because that would slow down the performance.*/

override fun onDraw(canvas: Canvas) {
        /*
         * Compute the new position of the ball, based on accelerometer  data and present time.
         */

        val now = System.currentTimeMillis()
        mBallView.update(mSensorX, mSensorY, now, mHorizontalBound, mVerticalBound)


        /*
         * Transform the canvas so that the coordinate system matches
         * the sensors coordinate system with the origin in the center
         * of the screen and the unit is the meter.
         */
        val x = mXOrigin + mBallView.mPosX * mMetersToPixelsX
        val y = mYOrigin - mBallView.mPosY * mMetersToPixelsX

        mBallView.setTranslationX(x)
        mBallView.setTranslationY(y)

        // and make sure to redraw asap
        invalidate()
    }



    //Start the sensor listeneer when the activi starts/resumes, and unregister when the activity
sotps/destroyed
      ..........

}
```

# Assignment: Colliding Balls

## In your kotlin View class

```
/*
* The ball holds its previous and current position, its
* acceleration. for added realism , the ball has its own friction coefficient.
*/
class BallView : View {
    var mPosX =  0.toFloat()
    var mPosY = 0.toFloat()
    var mVelX = 0.toFloat()
    var mVelY = 0.toFloat()
    var mLastT: Long = 0

   //Secondary constructor
    constructor(context: Context, simulationView: SimulationView, mDstWidth: Int, mDstHeight: Int) : super(context) {
         setBackgroundResource(R.drawable.ball)
        Log.d("tag", "set the ball in canvas")
        simulationView.addView(this, ViewGroup.LayoutParams(mDstWidth, mDstHeight))    }
   /*
     * Performs one iteration of the simulation,  updating the  position of all the ball
     */
    fun update(sx: Float, sy: Float, now: Long, mHorizontalBound: Float, mVerticalBound: Float) {
        // update the system's positions
        val dT = (now - mLastT).toFloat() / 1000f
        /** (1.0f / 1000000000.0f) */
        computePhysics(sx, sy, dT)
        mLastT = now
       resolveBoundEdges(mHorizontalBound, mVerticalBound)   }

    fun computePhysics(sx: Float, sy: Float, dT: Float) {
        val ax = -sx / 10
        val ay = -sy / 10
        mPosX += mVelX * dT + ax * dT * dT / 2
        mPosY += mVelY * dT + ay * dT * dT / 2
        mVelX += ax * dT
        mVelY += ay * dT
    }
```

# Assignment: Colliding Balls

In your kotlin View class

```
/*
     * Resolving horizontal and vertical constraints
     */
    fun resolveBoundEdges(mHorizontalBound: Float, mVerticalBound: Float) {
        val xmax = mHorizontalBound
        val ymax = mVerticalBound
        val x = mPosX
        val y = mPosY
        if (x > xmax) {
            .....
        } else if (x < -xmax) {
            .....
        }
        if (y > ymax) {
            .....
        } else if (y < -ymax) {
            .....
        }
    }


}
```

# Assignment: Face Detection

In your main activity:

1. Create the Face Detector

```
val detector = FaceDetector.Builder(applicationContext)
    .setTrackingEnabled(false)
    .setLandmarkType(FaceDetector.ALL_LANDMARKS)
    .setClassificationType(FaceDetector.ALL_CLASSIFICATIONS)
    .build()
```

2. Create a frame from the bitmap and run the face detection on the frame

```
val frame = Frame.Builder().setBitmap(bitmap).build()
val faces = detector.detect(frame)
```

# Assignment: Face Detection

In you View Class

- Receive the faces and the bitmap from the main activity
- Process the bitmap based on the landmarks
- Pain the result on a Canvas

```
class myViewClass(context: Context, attrs: AttributeSet) : View(context, attrs)
{

     private var mBitmap: Bitmap? = null
     private var mFaces: SparseArray<Face>? = null

          . . . . .
```

# Assignment: Face Detection

In you View Class

- Scale the Bitmap to the size of the Canvas

```
val viewWidth = canvas.width.toDouble()
val viewHeight = canvas.height.toDouble()
val imageWidth = mBitmap!!.width.toDouble()
val imageHeight = mBitmap!!.height.toDouble()
val scale = Math.min(viewWidth / imageWidth, viewHeight / imageHeight)
```

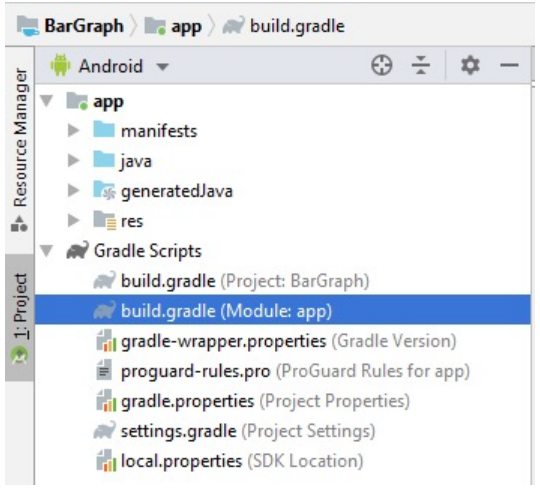
# Assignment: Face Detection

In you View Class

- Iterate over the faces, and recover the landmarks

```
for (i in 0 until mFaces!!.size()) {
    val face = mFaces!!.valueAt(i)
    var smileProb = face.isSmilingProbability
    for (landmark in face.Landmarks) {
        val cx = (landmark.position.x * scale).toInt()
        val cy = (landmark.position.y * scale).toInt()
        //Example of plotting a circle in the landmark positions
        canvas.drawCircle(cx.toFloat(), cy.toFloat(), 10f, paint)

    }
}
```

# Assignment: Face Detection




```
dependencies {
     . . . . .
    implementation 'com.google.android.gms:play-services-vision:9.4.0’
         . . . . .
}
```

# Your Questions