More Async Tools, Sensors

WorkManager

- WorkManager is used for “scheduling persistent work”
- Persistent means that things which are scheduled will run as appropriate even if the app is closed or the device is rebooted
- However, it can also be used for running immediate, but long running tasks in the background
- We can specify constraints about when tasks should be run such as only when battery is high or only when connected to wifi

WorkManager Basics

- The actual work we want to do goes in the `doWork()` method in classes that inherit from `Worker`
- These methods take `WorkerParameters` and return a `Result`
- We create `WorkRequest` objects which specify which worker to run, whether it's a one-shot job, or a periodic job, what to do if there's already an instance of that worker running, and other data
- We `enqueue` workRequests to start them off

Chaining Tasks

- WorkRequests can be chained to specify graphs of dependent work to be performed
- The simplest is a sequence of tasks which can be chained with a request's

  `then()` method which takes a WorkRequest to start after completion
- The `combine` method waits for multiple tasks to complete
- With these methods we specify the dependencies between our tasks and the WorkManager will automatically schedule the jobs for us

Example: Image Blurring App

Adapted from `This Codelab`

Sensors

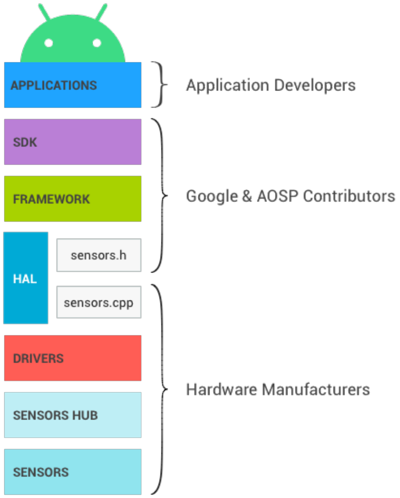
Outline

- Android devices have a ton of sensors in them that you can access
- Examples include accelerometer, gyroscope, compass, etc
- We'll look at how to access these in our apps

Sensors

- They measure physicical quantities: acceleration, light, temperature, angle to magnetic north, etc
- These are usually small electromechanical devices that convert some physical quantitity to a voltage/current
- The voltage/current is digitized using an “analog to digital converter” which gives us numbers
- The sensors themselves are often use pretty limited power

Sensor Stack



Sensor Stack

Physical/Virtual Sensors

- Often the low level sensors are too noisy to be used directly, or require some sort of calibration, etc
- People who understand the sensors at a low level can write code to “fix” the physical sensor measurements
- The “fixed” measurements are presented as sensors to the Framework, so as an App programmer, the interface to these “virtual” sensors is the same as the interface to the “physical” sensors

Sensor Categories

- Sensors are split into 3 categories:
  - Motion sensors: measure motion (indirectly) such as velocity/acceleration or rotation
  - Position sensors: An example of this would be the compass
  - Envionmental sensors: temperature, humidity, etc

What are we measuring

- Some stuff we measure is “scalars” like temperature, humidity, etc. Scalars are “one number” per measurement
- Other measurements are “vectors” likeposition, acceleration, etc which are “many numbers” per measurement
- Usually the vector components are measurements along different axes
- We should know what to expect based on the sensor we're reading from

Coordinate System

- We need to know what direction the axes are pointed in to make sense of vector measurements
- The point (0,0,0) is at the usually at the center of the screen
- The x axis is horizontal across the screen
- The y axis is vertical along the screen
- The z axis is into/out of the screen
- You may want to pin the app's orientation if you're using these sensors for consistency
- Some measurements put the origin at the center of the earth, so you may need to convert between systems

Sensor Values

- Sensors give 3 types of data:
  - Raw: this is sensor data that is directly collected from the sensor. Eg.

    Accelerometer, light sensors, proximity sensors.
  - Calibrated: the Android OS postprocesses raw sensor data to remove noise, drift, bias etc, and reports the postprocessed values. Eg. Step detector.
  - Fused: combined data from one or more sensors. Eg. Gravity sensor data is a combination of accelerometer and gyroscope data.

The API

- The “Sensor Framework” lives in the android.hardware package
- There are 4 main components:
  - SensorManager
  - Sensor
  - SensorEvent
  - SensorEventListener

SensorManager

- This is the Service that manages sensors
- You can get a list of available sensors from it
- You can register sensor event listeners through it
- Provides info about coordinate systems and important constants

Sensor

- This is the base class for all Sensors
- Provides lots of useful info about sensors such as:
  - Maximum range (range of values you might read)
  - Minimum delay
  - Name
  - Power usage
  - “Reporting Mode”
  - Resolution (precision)

SensorEvent + SensorEventListener

- These are how you actually get data from a sensor
- You register callbacks for the sensors your care about (probably `onSensorChanged()`, possibly others)
- That callback takes a `SensorEvent` with the measured data:
  - `values[]` — the actual data
  - `timestamp` — at nanosecond precision!
  - `accuracy` — (high, medium, or low)
  - `sensor` — the sensor

Reporting Modes

- There are different “reporting modes” that trigger callbacks differently
- Continuous: update at a constant frequency
- On-Change: only update if the measurement changes
- One-Shot: only measure once

Typical Workflow

- Get the `SensorManager` instance
- Use it to grab the `Sensor`s you want to use
- Register listeners for those sensors with the manager
- Get your measurements when the callbacks are triggered
- Unregister your listener — important so you're not running your callback hundreds of times a second when you don't need to!

Example: Listing Sensors

- We'll use the SensorManager to list available sensors and some info about them
- We'll use Jetpack Compose again for the UI... It's REALLY nice for stuff like this!
- In a real app, you'll want to search the list of sensors to find the one(s) you need before registerring callbacks

Example: Reading from ambient light sensor

- We'll just display values from the Ambient Light sensor
- This shows the process of actually reading from a sensor
- It's important to register/unregister our sensor callbacks in `onPause` and `onResume` so we keep waking up our app to read the sensor when the view we're updating isn't visible

Running on physical hardware

- Step 1: enable “developer options” by finding the “build number” in your settings menu and clicking a bunch of times (it should tell you how many more times to click it)
- Step 2: enable “USB Debugging” in settings
- Step 3: connect via USB cable, and after some permisions stuff on your mac + phone, you should see it in the Android Studio “Device Manager”
- Step 4: Pick your physical device, then hit the play button