twitch.tv/dancojocar
youtube.com/dancojocar

Lecture #8
System Services & Sensors

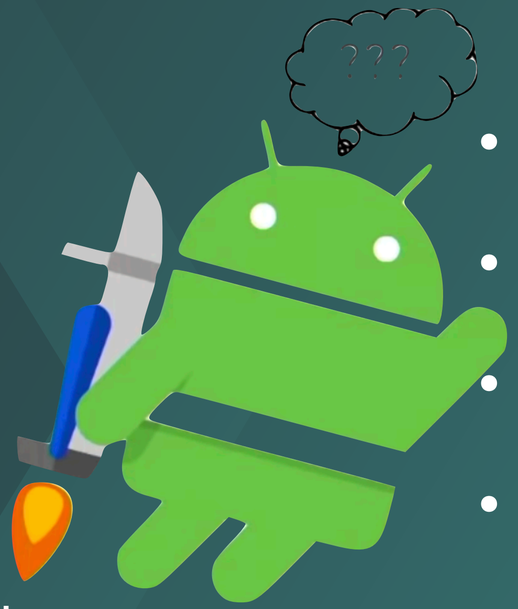
Mobile Applications
Fall 2023

Background Tasks

- Sending logs or tracking user progress.
- Upload images, videos or session data.
- Synching data.
- Processing data.

Options

- Threads
- Executors
- Services
- AsyncTasks
- Handlers and Loopers
- Coroutines



- Jobs (API 21+)
- GcmNetworkManager
- SyncAdapters
- Loaders
- AlarmManager
- WorkManager

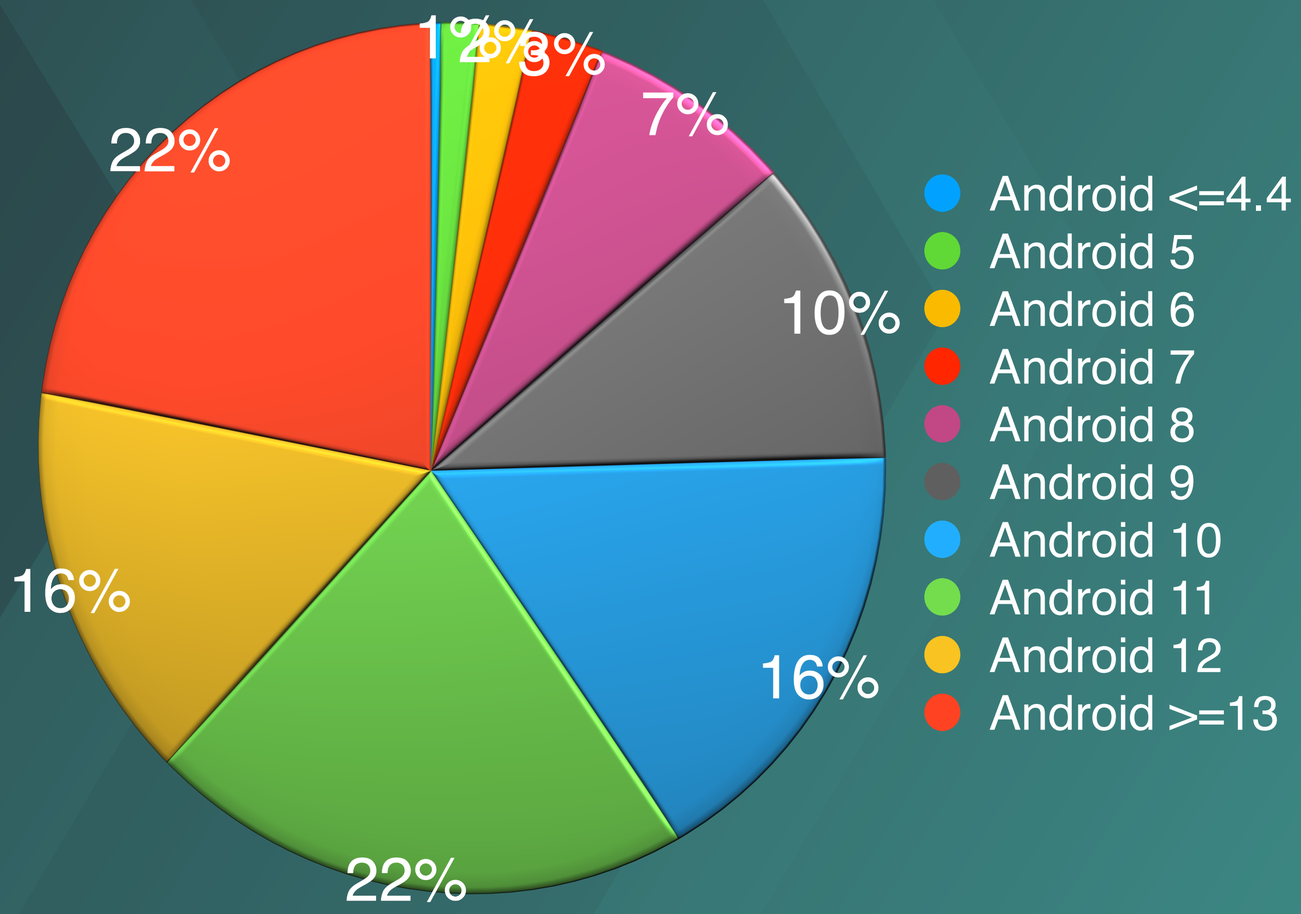
Battery Optimizations

- Doze mode
- App standby

- Limited implicit broadcasts

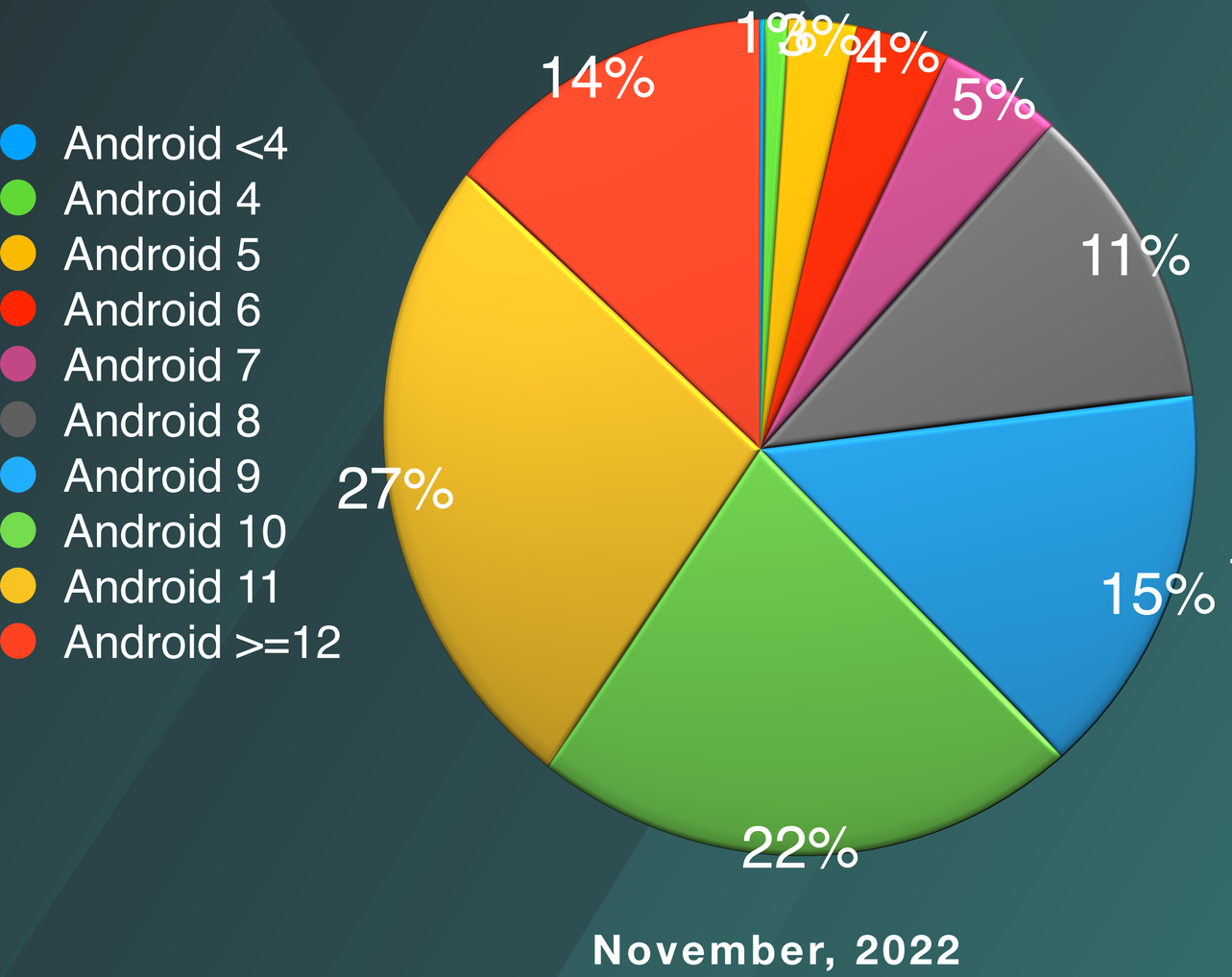
- Release cached wakelocks
- Background service limitations
- App standby buckets
- Background restricted apps

Compatibility

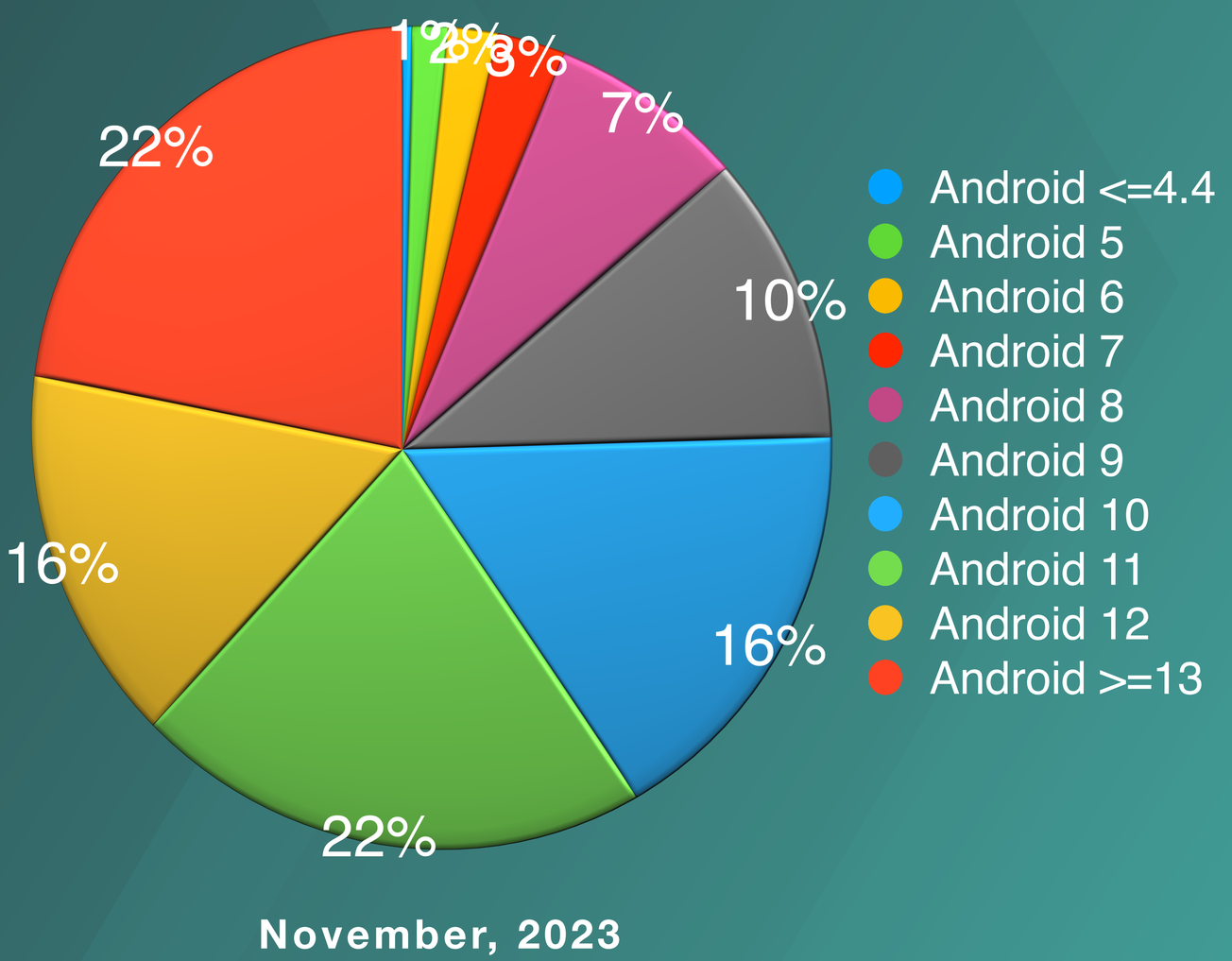


November, 2023

Compatibility

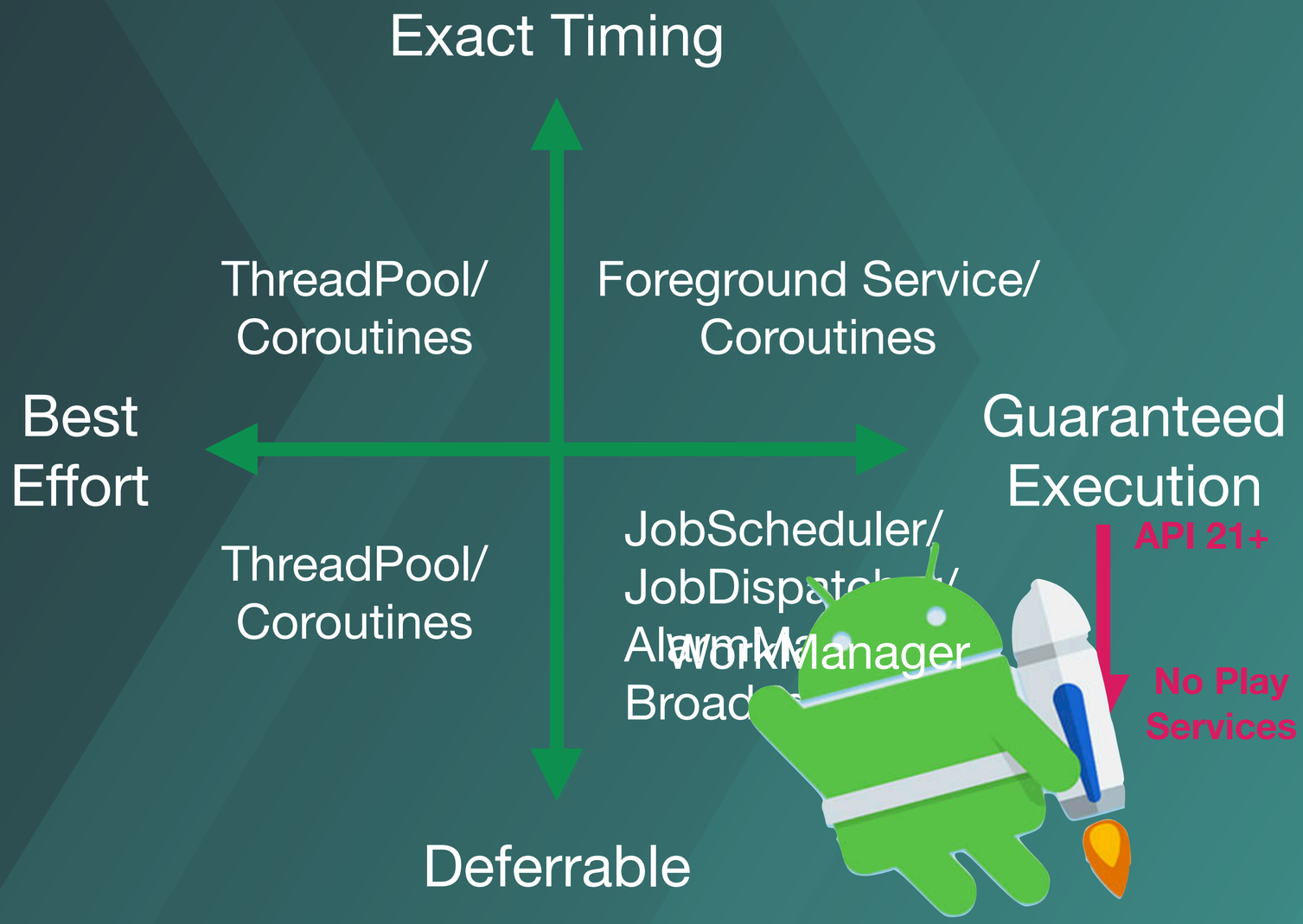


November, 2022



November, 2023

Requirements

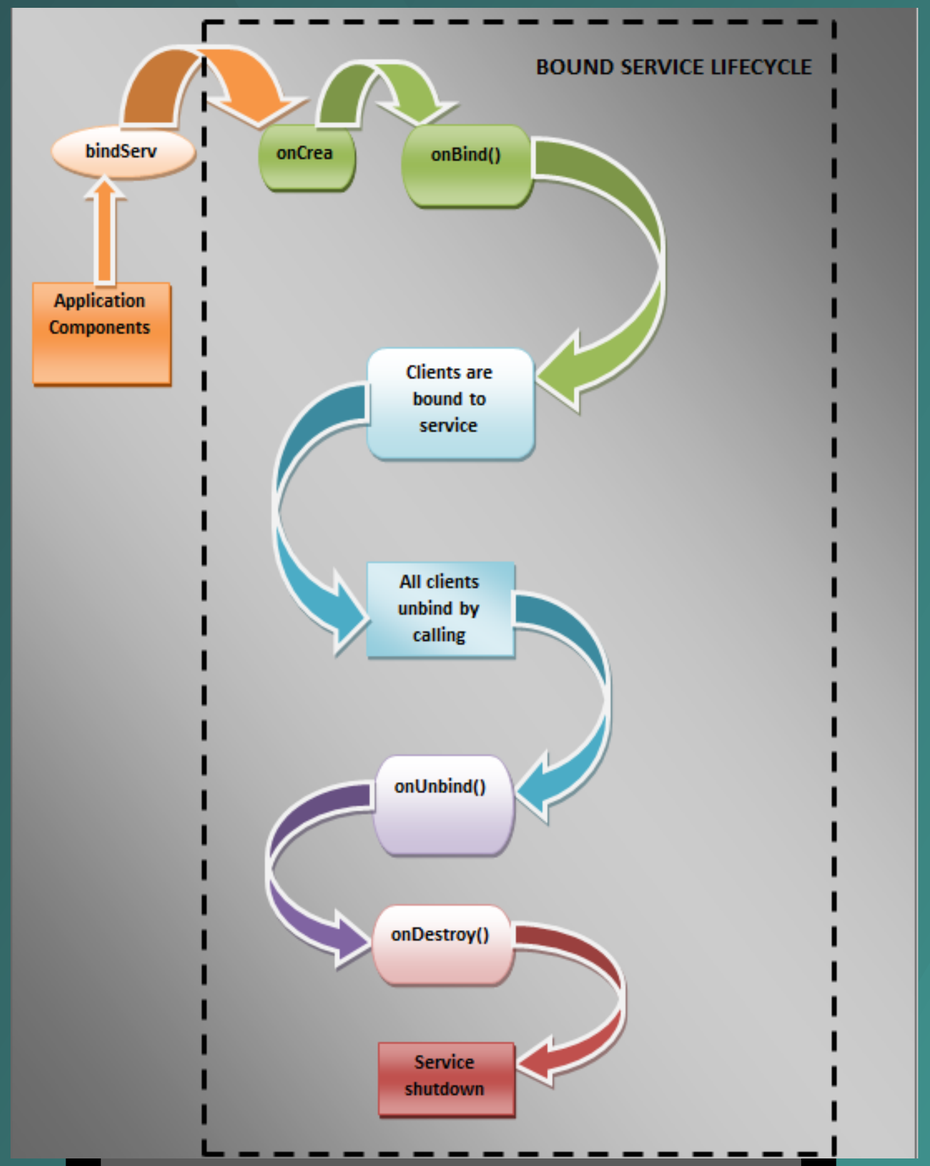


Services

- Perform long-running operations in the background.
- It does not provide a user interface.
- Continues to run even is the user switches to another application.
- Eg.:
  - Network transactions.
  - Play music.
  - Perform I/O.
  - Use a content provider.

Services

- Types:
  - Foreground
  - Background
  - Bound



Basics

- onStartCommand()
- onBind()
- onCreate()
- onDestroy()

The system invokes this method by calling `startService()` when another component (such as an activity) requests that the service be started.

The system invokes this method by calling `bindService()` when another component wants to bind with the service (such as to perform RPC).

The system invokes this method to perform one-time setup procedures when the service is first created.

The system invokes this method when the service is no longer used and is being destroyed.

In your implementation of this method, you must provide an interface that clients use to communicate with the service by returning an `IBinder`.

You must always implement this method; however, if you don't want to allow binding, you need to implement this method.

In your implementation of this method, you must provide an interface that clients use to communicate with the service by returning an `IBinder`.

If you only want to provide binding, you don't should return null.

You must always implement this method; however, if you don't want to allow binding, you need to implement this method.

Your service should implement this to clean up any resources such as threads, registered listeners, or receivers.

If you implement this, it is your responsibility to stop the service when its work is complete by calling `stopSelf()` or `stopService()`.

This is the method that the system calls when it is no longer used and is being destroyed.

If you implement this, it is your responsibility to stop the service when its work is complete by calling `stopSelf()` or `stopService()`.

This is the method that the system calls when it is no longer used and is being destroyed.

Your service should implement this to clean up any resources such as threads, registered listeners, or receivers.

This is the method that the system calls when it is no longer used and is being destroyed.

The system invokes this method by calling `startService()` when another component (such as an activity) requests that the service be started.

The system invokes this method by calling `bindService()` when another component wants to bind with the service (such as to perform RPC).

The system invokes this method to perform one-time setup procedures when the service is first created.

The system invokes this method when the service is no longer used and is being destroyed.

In your implementation of this method, you must provide an interface that clients use to communicate with the service by returning an `IBinder`.

You must always implement this method; however, if you don't want to allow binding, you should return null.

You must always implement this method; however, if you don't want to allow binding, you need to implement this method.

Your service should implement this to clean up any resources such as threads, registered listeners, or receivers.

If you implement this, it is your responsibility to stop the service when its work is complete by calling `stopSelf()` or `stopService()`.

This is the method that the system calls when it is no longer used and is being destroyed.

developer.android.com/guide/components/services

Declaring a service in the manifest

```
<manifest ... >
  ...
  <application ... >
      <service android:name=".ExampleService" />
      ...android:description="Service Description"
  </application>
</manifest>
  </application>
</manifest>
```

developer.android.com/guide/topics/manifest/service-element

Creating a Service

- Service
  - Base class for all services.
  - Create and manage a new thread on your own.
- IntendService
  - Subclass of Service.
  - Uses a worker thread.
  - Implement onHandleIntent()

```
/**
 * A constructor is required
 * with a name for the worker thread.
 */
class HelloIntentService() {
    IntentService(name)
    private var mServiceLooper: Looper? = null
    private var mServiceHandler: ServiceHandler? = null
    /**
     * The IntentService calls this method
     * Handler that receives messages from the thread
     */
    private inner class ServiceHandler(looper: Looper) : Handler(looper) {
        override fun handleMessage(msg: Message) {
            // Normally we would do some work here, like download a file.
            // For our sample, we just sleep for 5 seconds.
            try {
                Thread.sleep(5000)
                Thread.sleep(5000)
            } catch (e: InterruptedException) {
                // Restore interrupt status.
                Thread.currentThread().interrupt()
                Thread.currentThread().interrupt()
            }// Stop the service using the startId, so that we don't stop
            // the service in the middle of handling another job
            stopSelf(msg.arg1)
        }
    }
```

Service Management

- Starting a service
- Stopping a service

```
Intent(this, HelloService::class.java).also {
  intent -> startService(intent)
}
```

```
stopService()
```

```
stopSelf()
```

Notify the User

- Toast Notifications
- Status Bar Notifications

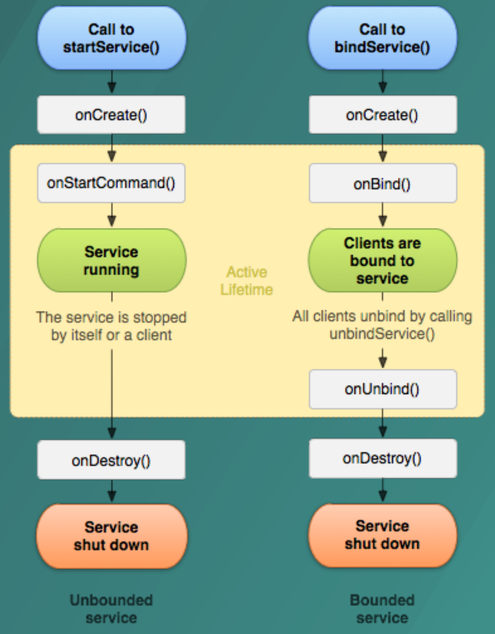
```
val pendingIntent: PendingIntent =
 Intent(this, ExampleActivity::class.java).let { notificationIntent ->
  PendingIntent.getActivity(this, 0, notificationIntent, 0)
 }
```

```
  val text = "Hello toast!"
  val duration = Toast.LENGTH_SHORT
val notification: Notification = Notification.Builder(this,
 CHANNEL_DEFAULT_IMPORTANCE)
  val toast = Toast.makeText(applicationContext, text, duration)
  toast.show()
 .setContentTitle(getText(R.string.notification_title))
 .setContentText(getText(R.string.notification_message))
 .setSmallIcon(R.drawable.icon)
 .setContentIntent(pendingIntent)
 .setTicker(getText(R.string.ticker_text))
 .build()

startForeground(ONGOING_NOTIFICATION_ID, notification)
```

Service Lifecycle

```
class ExampleService : Service() {
 private var mStartMode: Int = 0 // how to behave if the service is killed
 private var mBinder: IBinder? = null // interface for clients that bind
 private var mAllowRebind: Boolean = false // whether onRebind should be used

 override fun onCreate() {
  // The service is being created
 }
 override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
  // The service is starting, due to a call to startService()
  return mStartMode
 }
 override fun onBind(intent: Intent): IBinder? {
  // A client is binding to the service with bindService()
  return mBinder
 }
 override fun onUnbind(intent: Intent): Boolean {
  // All clients have unbound with unbindService()
  return mAllowRebind
 }
 override fun onRebind(intent: Intent) {
  // A client is binding to the service with bindService(),
  // after onUnbind() has already been called
 }
 override fun onDestroy() {
  // The service is no longer used and is being destroyed
 }
}
```

DEMO



Alarm Manager

- Alarm types:
  - ELAPSED_REALTIME
  - ELAPSED_REALTIME_WAKEUP
  - RTC
  - RTC_WAKEUP

```
alarmMgr?.setInexactRepeating(
        AlarmManager.ELAPSED_REALTIME_WAKEUP,
        SystemClock.elapsedRealtime() + AlarmManager.INTERVAL_HALF_HOUR,
        AlarmManager.INTERVAL_HALF_HOUR,
        alarmIntent
)
```

```
// Cancel the alarm.
alarmMgr?.cancel(alarmIntent)
```

developer.android.com/training/scheduling/alarms

Alarm at 14:00


```
// Set the alarm to start at approximately 2:00 p.m.
val calendar: Calendar = Calendar.getInstance().apply {
    timeInMillis = System.currentTimeMillis()
    set(Calendar.HOUR_OF_DAY, 14)
}

// With setInexactRepeating(), you have to use one of the AlarmManager interval
// constants--in this case, AlarmManager.INTERVAL_DAY.
alarmMgr?.setInexactRepeating(
        AlarmManager.RTC_WAKEUP,
        calendar.timeInMillis,
        AlarmManager.INTERVAL_DAY,
        alarmIntent
)
```

JobScheduler

- Register the Service

```
<service
    android:name=".MyJobService"
    android:permission="android.permission.BIND_JOB_SERVICE"
    android:exported="true"/>
```

- Schedule a Job

The Job

```
val builder = JobInfo.Builder(jobId++, serviceComponent)
builder.setMinimumLatency(delay.toLong() * TimeUnit.SECONDS.toMillis(1))
builder.setOverrideDeadline(deadline.toLong() * TimeUnit.SECONDS.toMillis(1))
builder.setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY)
builder.run {
  setRequiresDeviceIdle(requiresIdleCheckbox.isChecked)
  setRequiresCharging(requiresChargingCheckBox.isChecked)
  setExtras(extras)
}
(getSystemService(Context.JOB_SCHEDULER_SERVICE) as JobScheduler).schedule(builder.build())
```

Define The Job

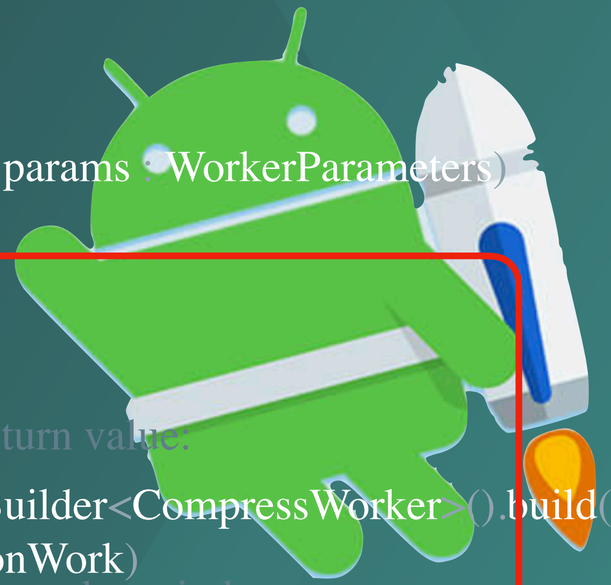
DEMO

```
class MyJobService : JobService() {
  override fun onStartCommand(intent: Intent, flags: Int, startId: Int): Int {
   activityMessenger = intent.getParcelableExtra(MESSENGER_INTENT_KEY)
   return Service.START_NOT_STICKY
  }

  override fun onStartJob(params: JobParameters): Boolean {
   // The work that this service "does"
   // ...
   // Return true as there's more work to be done with this job.
   return true
  }

  override fun onStopJob(params: JobParameters): Boolean {
   // Stop tracking these job parameters, as we've 'finished' executing.
   // ...
   // Return false to drop the job.
   return false
  }
}
```

developer.android.com/topic/performance/scheduling

WorkManager

- Core Classes:
  - Worker
  - WorkRequest
    - OneTimeWorkRequest
    - PeriodicTimeWorkRequest



```
class CompressWorker(context : Context, params : WorkerParameters)
    : Worker(context, params) {
    override fun doWork(): Result {
        // The actual work:
        myCompress()
        // Indicate success or failure with your return value:
        return Result.SUCCESS
        // Returning:
        //  - RETRY tells WorkManager to try this task again later
        //  - FAILURE says not to try again.
    }
}
```

```
val compressionWork = OneTimeWorkRequestBuilder<CompressWorker>().build()
WorkManager.getInstance().enqueue(compressionWork)
```

Runs on a background thread

Constraints

```
// Create a Constraints object that defines when the task should run
val myConstraints = Constraints.Builder()
 .setRequiresDeviceIdle(true)
 .setRequiresCharging(true)
 // ...
 .build()

// then create a OneTimeWorkRequest that uses those constraints
val compressionWork = OneTimeWorkRequestBuilder<CompressWorker>()
 .setConstraints(myConstraints)
 .build()
```

Cancel the task

```
 val compressionWorkId:UUID = compressionWork.getId()
 WorkManager.getInstance().cancelWorkById(compressionWorkId)
```

developer.android.com/topic/libraries/architecture/workmanager/basics

Chained Tasks

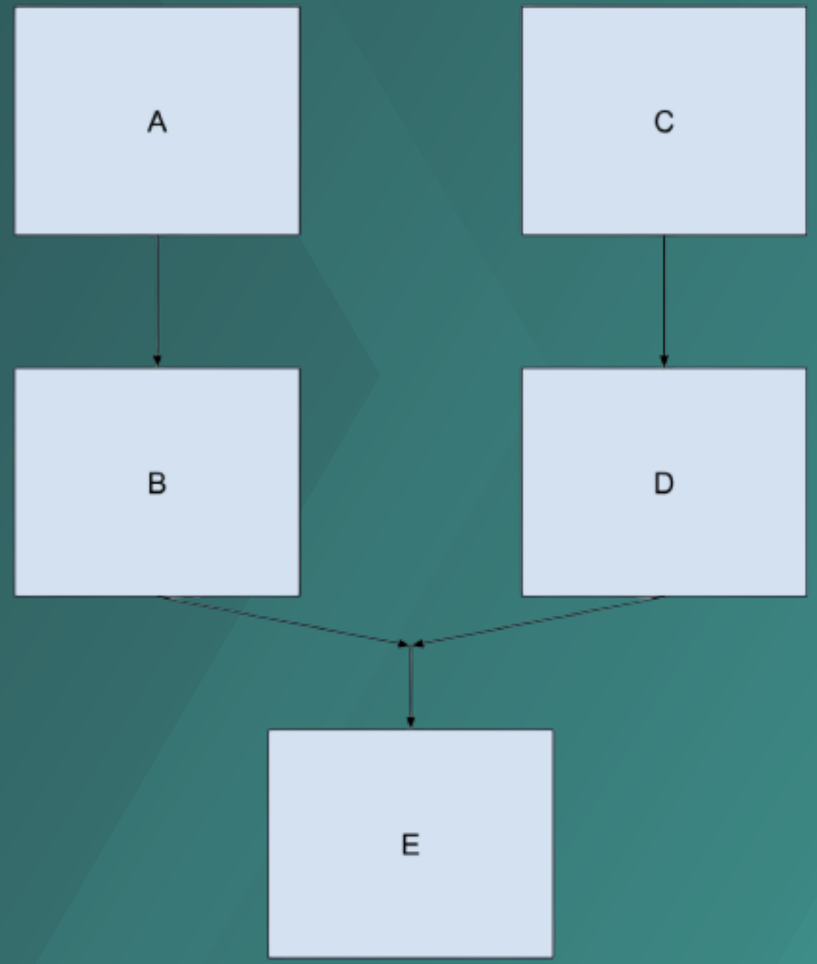
```
WorkManager.getInstance()
 .beginWith(workA)
   // Note: WorkManager.beginWith() returns a
   // WorkContinuation object; the following calls are
   // to WorkContinuation methods
   .then(workB) // FYI, then() returns a new WorkContinuation instance
   // First, run all the A tasks (in parallel):
   .beginWith(workA1, workA2, workA3)
   // ...when all A tasks are finished, run the single B task:
   .then(workB)
   // ...then run the C tasks (in any order):
   .then(workC1, workC2)
   .enqueue()
```


```
WorkManager.getInstance()
 .then(workB)
 .then(workC)
 .enqueue()
```

Chained tasks

DEMO

```
val chain1 = WorkManager.getInstance()
 .beginWith(workA)
 .then(workB)
val chain2 = WorkManager.getInstance()
 .beginWith(workC)
 .then(workD)
val chain3 = WorkContinuation
 .combine(chain1, chain2)
 .then(workE)
chain3.enqueue()
```



developer.android.com/topic/libraries/architecture/workmanager/advanced

Sensors

- Motion
- Position
- Environment

These sensors measure acceleration forces and rotational forces along three axes.

These sensors measure the physical position of a device.

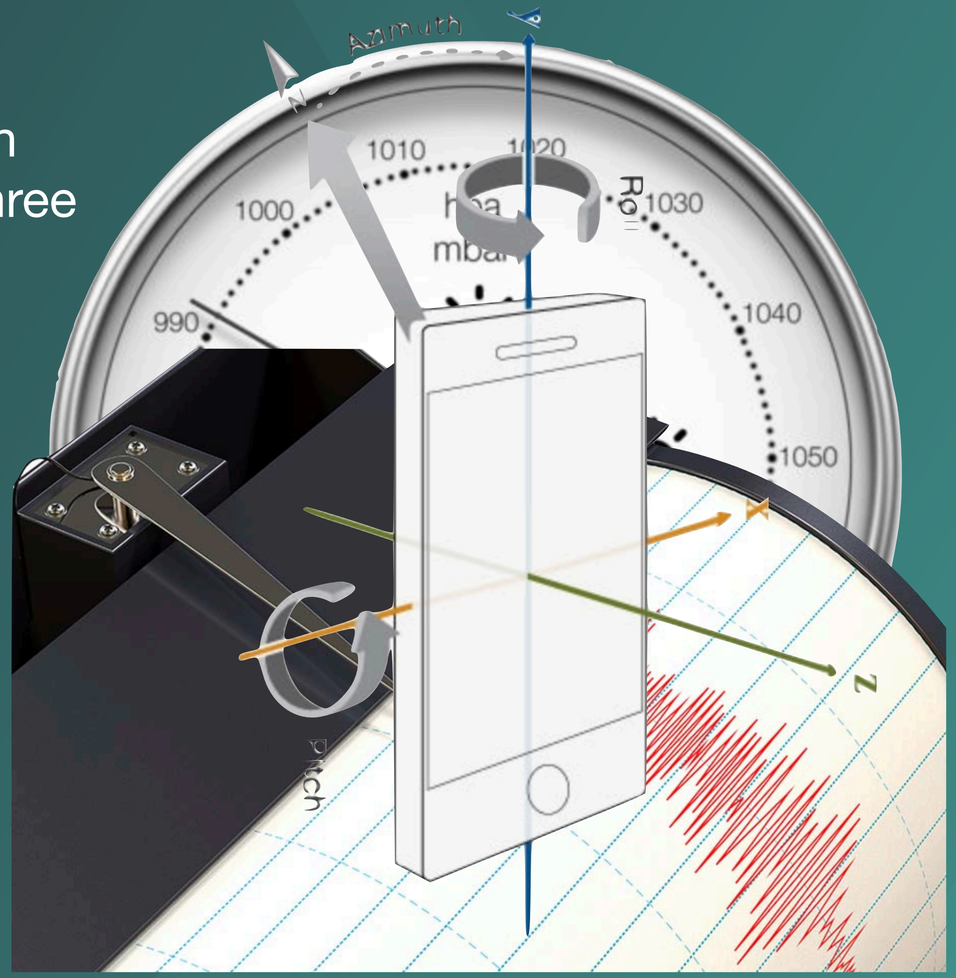
This category includes:

- Accelerometers.
- Gravity sensors.
- Gyroscopes.
- Rotational vector sensors.

These sensors measure various environmental parameters.

This category includes:

- Barometers.
- Photometers.
- Thermometers.



Framework

- SensorManager
- Sensor
- SensorEvent
- SensorEventListener

```
private lateinit var mSensorManager: SensorManager
...
mSensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
if (mSensorManager.getDefaultSensor(Sensor.TYPE_GRAVITY) != null) {
    val gravSensors: List<Sensor> =
} elmSensorManager.getSensorList(Sensor.TYPE_GRAVITY)
    // Use the version 3 gravity sensor.
} mSensor = gravSensors.firstOrNull {
    it.vendor.contains("Google LLC") && it.version == 3
}
}
class SensorActivity : Activity(), SensorEventListener {
private lateinit var mSensorManager: SensorManager
private var mLight: Sensor? = null
mSensor = if (mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)
override fun onCreate(savedInstanceState: Bundle?) {
    mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)
setContentView(R.layout.main)
// Sorry, there are no accelerometers on your device.
mSensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
mLight = mSensorManager.getDefaultSensor(Sensor.TYPE_LIGHT)
}
```

Motion

Sensor	Sensor event data	Description	Units of measure
TYPE_ACCELEROMETER	SensorEvent.values[0]	Acceleration force along the x axis (including gravity).	m/s^2
	SensorEvent.values[1]	Acceleration force along the y axis (including gravity).	
	SensorEvent.values[2]	Acceleration force along the z axis (including gravity).	
TYPE_ACCELEROMETER_UNCALIBRATED	SensorEvent.values[0]	Measured acceleration along the X axis without any bias compensation.	m/s^2
	SensorEvent.values[1]	Measured acceleration along the Y axis without any bias compensation.	
	SensorEvent.values[2]	Measured acceleration along the Z axis without any bias compensation.	
	SensorEvent.values[3]	Measured acceleration along the X axis with estimated bias compensation.	
	SensorEvent.values[4]	Measured acceleration along the Y axis with estimated bias compensation.	
	SensorEvent.values[5]	Measured acceleration along the Z axis with estimated bias compensation.	

Position

Sensor	Sensor event data	Description	Units of measure
TYPE_GAME_ROTATION_VECTOR	SensorEvent.values[0]	Rotation vector component along the x axis (x * sin(θ/2)).	Unitless
	SensorEvent.values[1]	Rotation vector component along the y axis (y * sin(θ/2)).	
	SensorEvent.values[2]	Rotation vector component along the z axis (z * sin(θ/2)).	
TYPE_GEOMAGNETIC_ROTATION_VECTOR	SensorEvent.values[0]	Rotation vector component along the x axis (x * sin(θ/2)).	Unitless
	SensorEvent.values[1]	Rotation vector component along the y axis (y * sin(θ/2)).	
	SensorEvent.values[2]	Rotation vector component along the z axis (z * sin(θ/2)).	
TYPE_MAGNETIC_FIELD	SensorEvent.values[0]	Geomagnetic field strength along the x axis.	μT
	SensorEvent.values[1]	Geomagnetic field strength along the y axis.	
	SensorEvent.values[2]	Geomagnetic field strength along the z axis.	
TYPE_MAGNETIC_FIELD_UNCALIBRATED	SensorEvent.	Geomagnetic field strength (without hard	μT

Environment

DEMO

Sensor	Sensor event data	Units of measure	Data description
TYPE_AMBIENT_TEMPERATURE	event.values[0]	°C	Ambient air temperature.
TYPE_LIGHT	event.values[0]	lx	Illuminance.
TYPE_PRESSURE	event.values[0]	hPa or mbar	Ambient air pressure.
TYPE_RELATIVE_HUMIDITY	event.values[0]	%	Ambient relative humidity.
TYPE_TEMPERATURE	event.values[0]	°C	Device temperature.[1]

★ [1] Implementations vary from device to device. This sensor was deprecated in Android 4.0 (API Level 14).

developer.android.com/guide/topics/sensors/sensors_environment

Lecture outcomes

- Use existing system services.
- Define custom services.
- Understand the user notifications API.
- Consume data from sensors.