COM4510/6510
Software Development for Mobile Devices

Lecture 8: Services, Threads and Locations

Dr Po Yang
The University of Sheffield
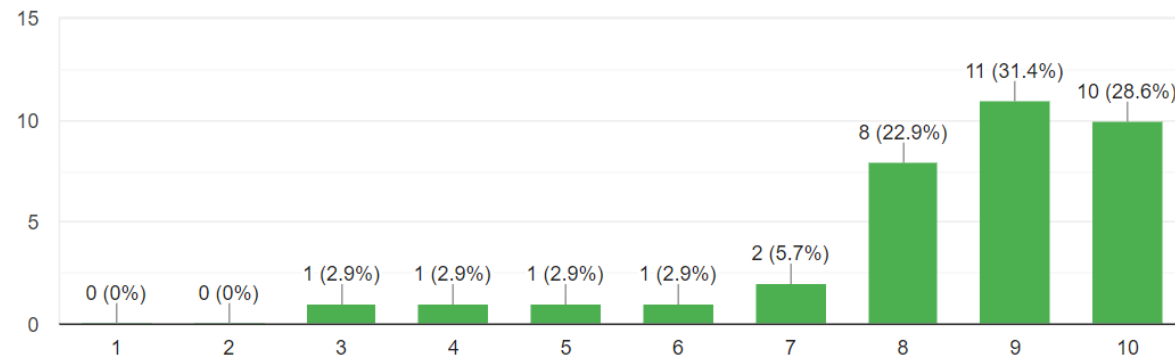po.yang@sheffield.ac.uk

Lecture Overview

- Part 1:
  - Services and Threads
  - Life-Cycle of Android Services
- Part 2:
  - Location in the background
  - Tracking locations in the background
  - Sensing in the background
- Lab tutorial :
  - Maps, Locations and Services
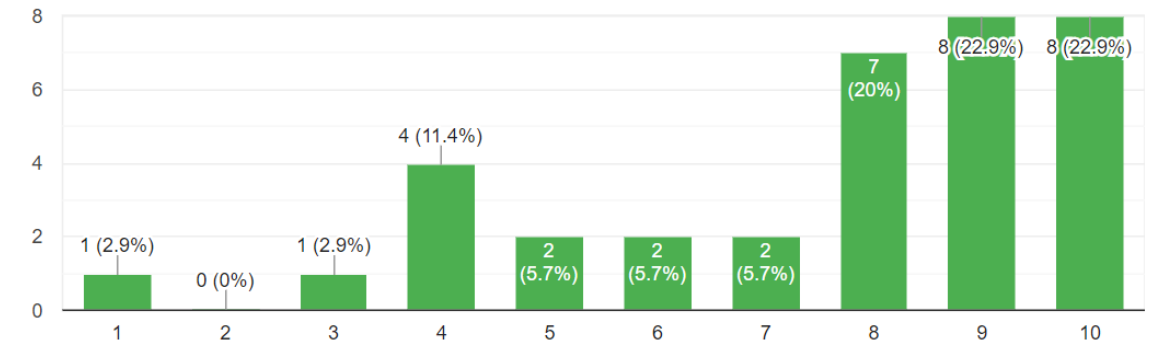
Response

- Good things

Is the teaching content relevant and up to date?
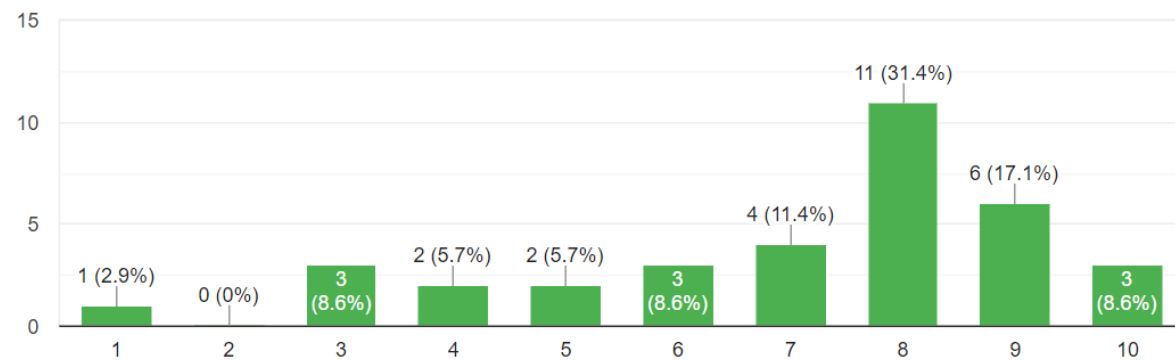
35 responses

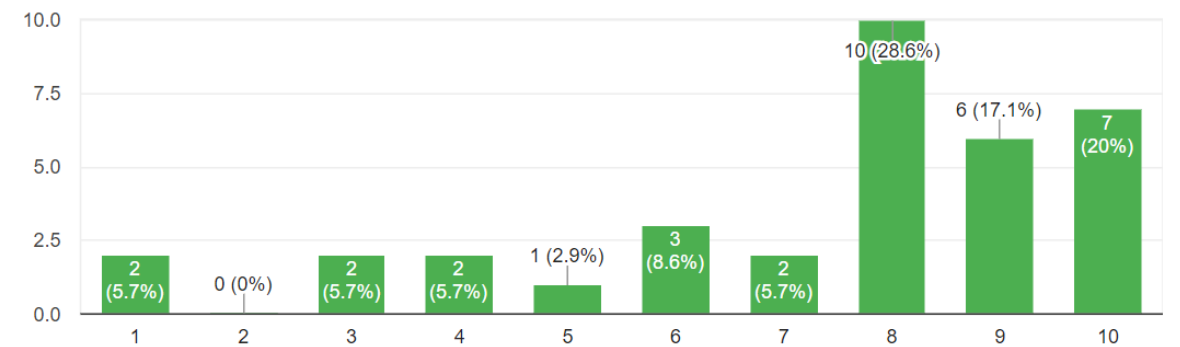How engaging were the instructors?

35 responses

How clear and easy to understand is the face-to-face lectures

35 responses

How would you rate your instructor's delivery of the material?

35 responses

Response

- Need improvements in running labs
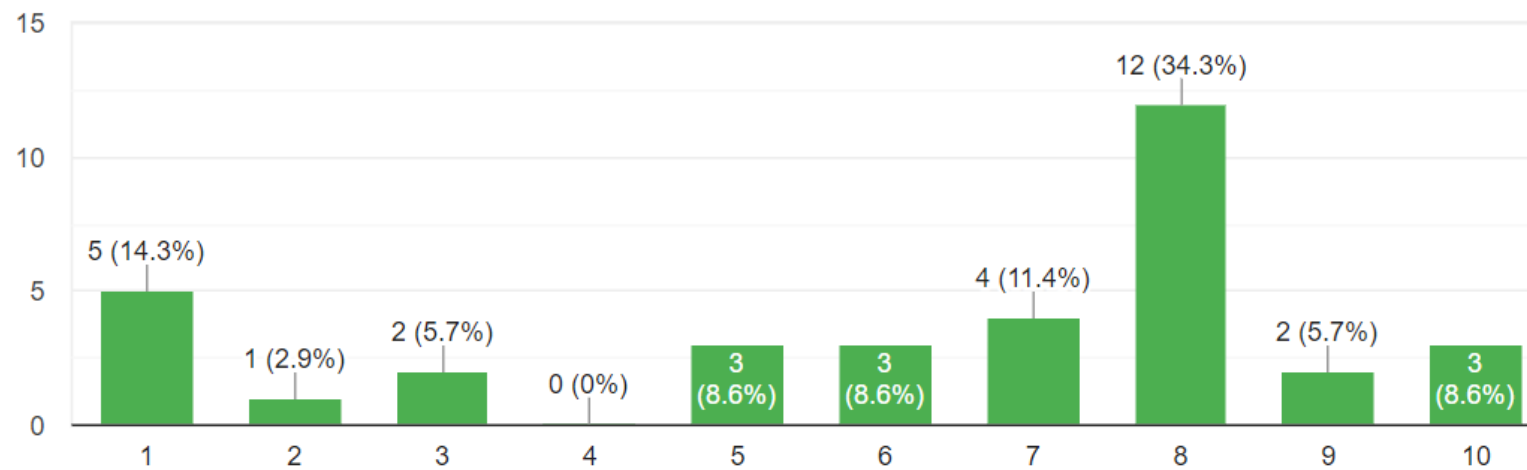
How useful/helpful are the face-to-face labs

35 responses

How useful/helpful are the online labs

35 responses
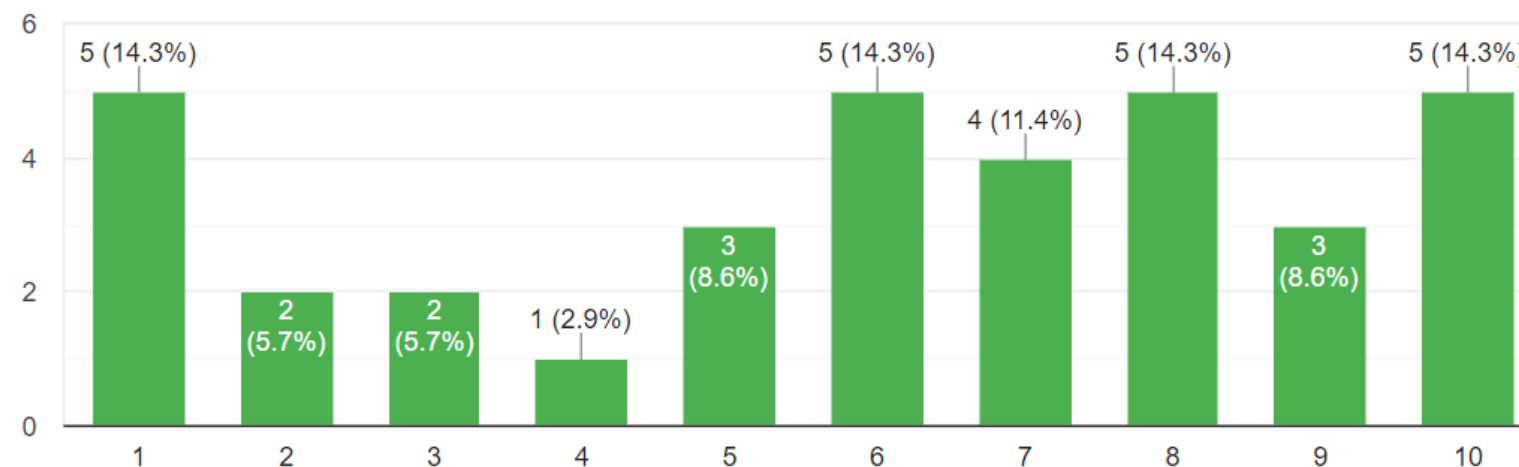
Some Feedback

- Labs should not be a mini lecture

- More interactive labs

- Google materials and tutorials are more useful

- Assessment (Teamwork and individual)

- Struggling to catch up after missing first few labs

- Software was not working in the Lab PCs

Actions

- Mini lecture will be no more than 5 minutes

- Breakout rooms in online lab

- One-to-one drop-off sessions

- Talk with me about your issues on Assessment (Teamwork and individual)

- Push GTAs to help a lot

Service

- A **Service** is an application component that can perform **long-running operations** in the background, and it does not provide a **user interface**

- Another application component **can start** a service, and it **continues to run in the background** even if the user switches to another application.

https://developer.android.com/guide/components/services

Service

- **Some common uses for a Service include:**

- **Downloading or uploading data** from/to a network in the background even if the app is closed

- **Saving data to a database** without crashing if the user leaves the app

- Running some other kind of long-running, **"background" task** even after the app is closed, such as playing music!

https://developer.android.com/guide/components/services

Threads and Processes

- **Concurrency** is the process by which we have multiple *processes* running at the same time.

- Computers are really good at *multitasking*

- These "tasks" are divided up into two types:

The shaded areas indicate when each thread is executing
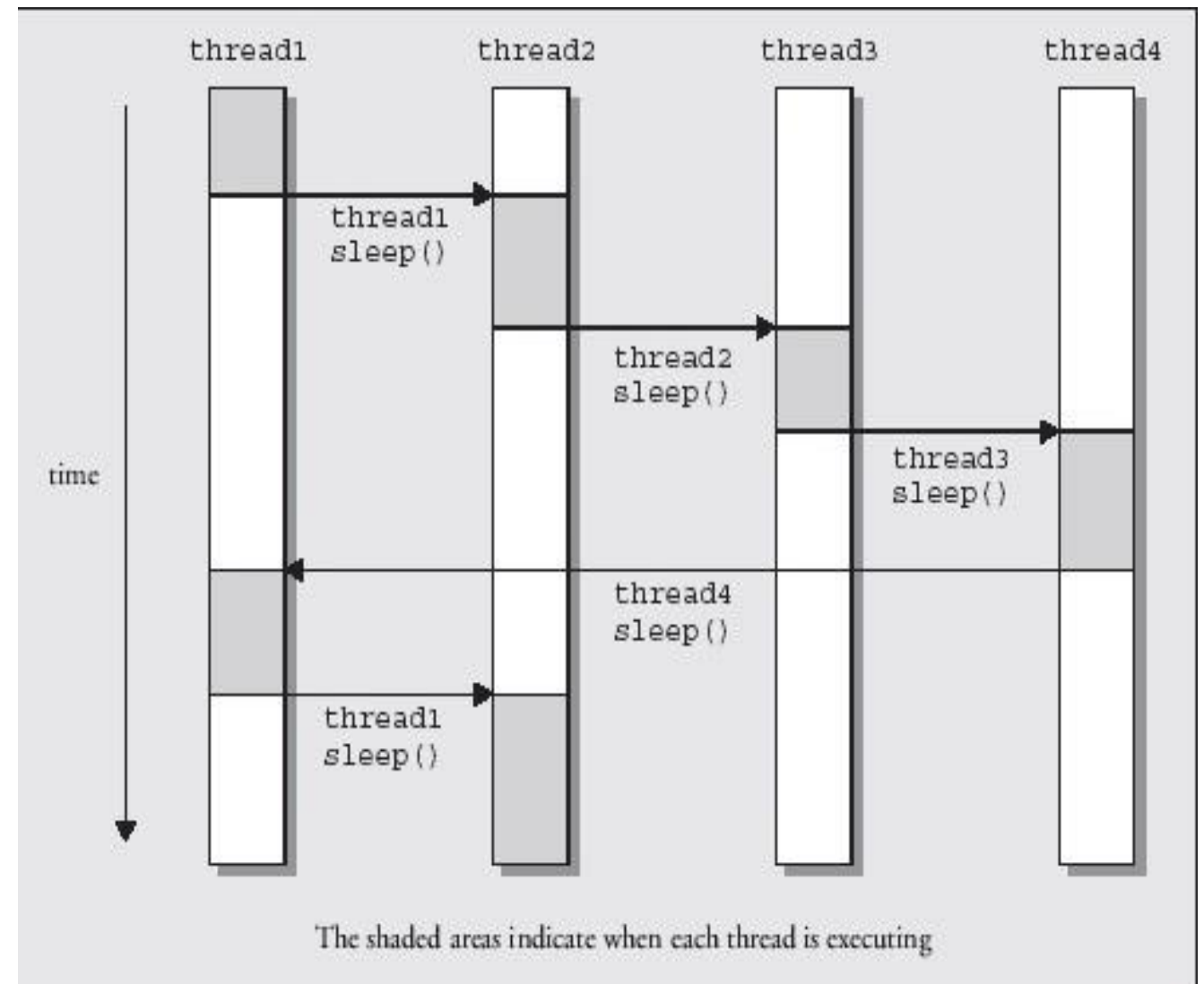
Processes: self-contained execution environment; complete, private set of basic run-time resources; its own memory space.

Threads are called lightweight processes.

https://info448.github.io/services.html

Android Threads

- Android apps run by default on the **Main Thread (UI Thread)**

  - It is in charge of all user interactions

  - Handling button presses, scrolls, drags, etc.

- Threads are a way that we can break up a single application or process into little "**sub-process**" that can be run **simultaneously**

- Within a single thread, all method calls are **synchronous**

- However, **long, drawn-out processes** cause other tasks to have to wait, lead to **ANR.**

Executing Computations

- Not do any **heavy computation** on the **UI Thread**

- Move any slow code onto **a background thread**

- We have already seen **AsyncTask** as a way to perform background tasks

- There are three ways of doing **long term computations**

  - **AsyncTask** for short-ish background computation

  - **Threads**: for parallel computation

  - **Services**:
    - for active computation on the UI thread (Service) or a separate thread (IntentService)
    - basically activities without visible UI

© Po Yang, University of Sheffield

- **A Service is not a separate process**.

  - The Service object itself does not imply it is running in its own process; unless otherwise specified, it runs in the same process as the application it is part of.

- **A Service is not a thread**.

  - It is not a means itself to do work off of the main thread (to avoid Application Not Responding errors)

- Thus a Service itself is very simple, **two main features**:

  - A facility for the application to tell the system *about* something it wants to be **doing in the background**

  - A facility for an application to **expose some of its functionality to other** applications.

Types of Services

https://www.geeksforgeeks.org/services-in-android-with-example/

13
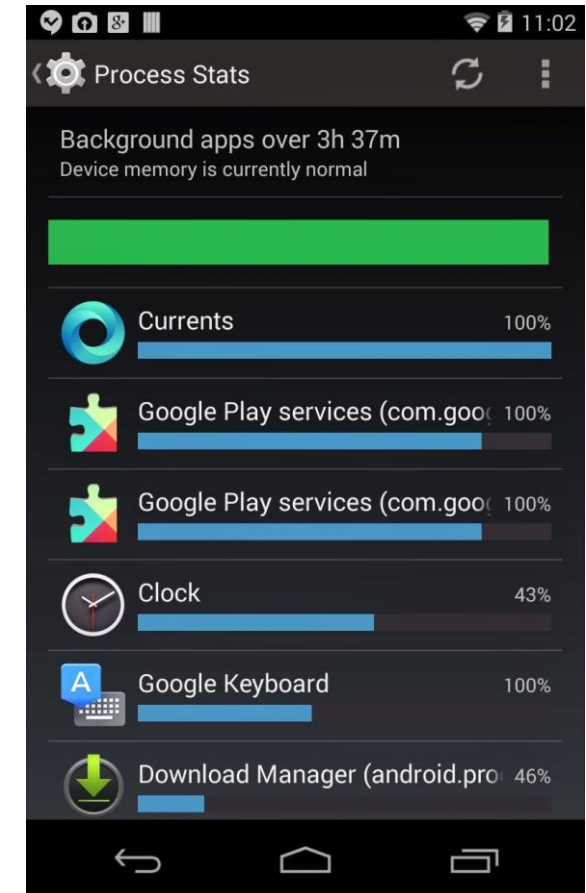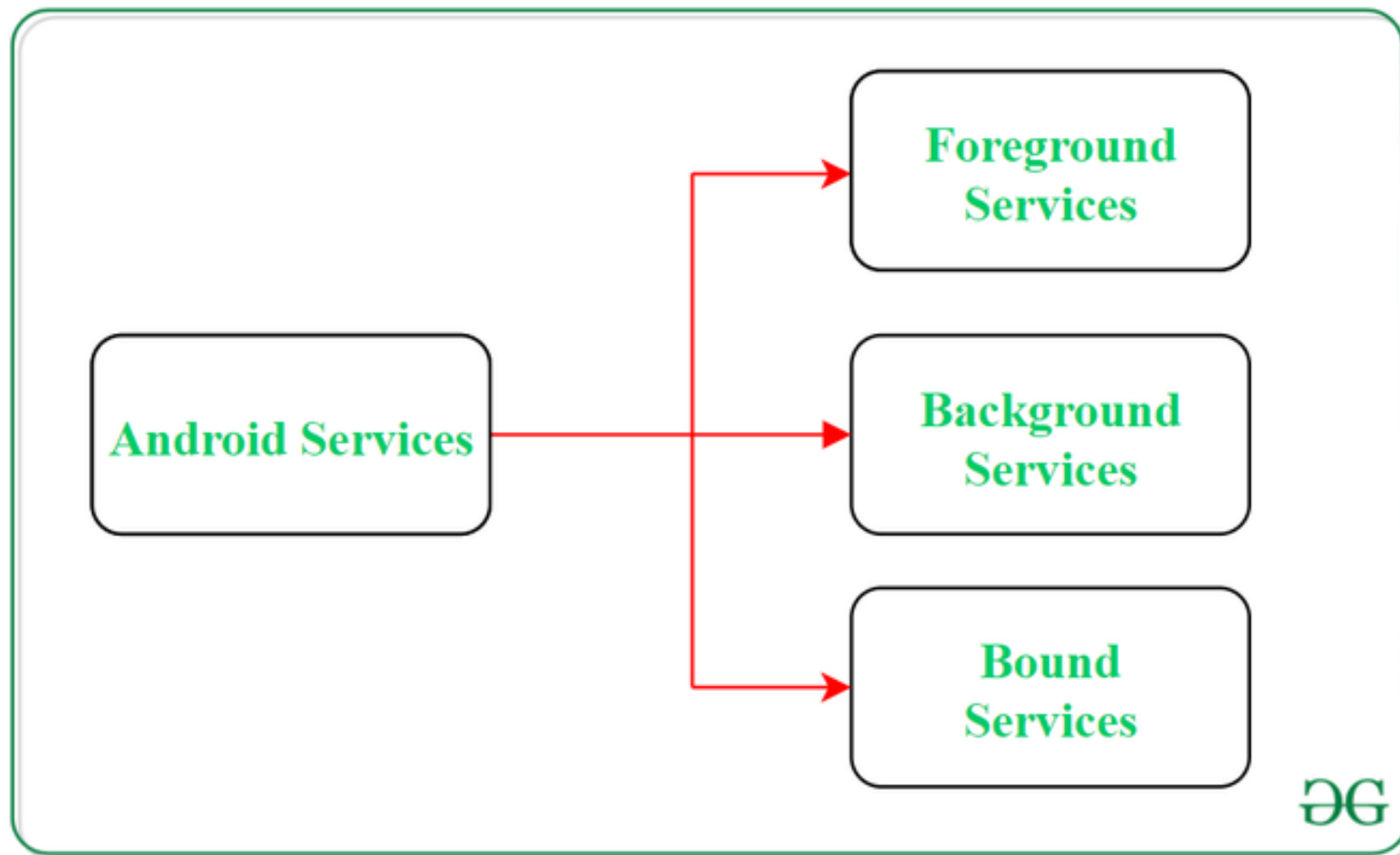
- **Foreground**
  - performs some operation that is noticeable to the user
    - e.g. **audio app** would use a foreground service to play an audio track.
  - must display a **status bar icon**
  - continues running even when the **user isn't interacting with the app**

  Currently the only **safe way to do long running** background computation

- **Background**
  - performs an operation that isn't directly noticed by the user
    - if an app used a service to compact its storage, that would usually be a background service.
  - Increasingly **unreliable** in order to **save battery**
    - location tracking is only available a few times an hour when in background

- **Bound**

  - when an application component binds to it by calling **bindService()**

  - offers a **client-server interface** that allows components to interact with the service, send requests, receive results, and

  - A bound service runs only as long as another application component is bound to it

Background Services

⭐ Note: If your app targets API level 26 or higher, the system imposes restrictions on running background services when the app itself isn't in the foreground. In most cases like this, your app should use a scheduled job instead.

The system **distinguishes between** *foreground* **and** *background* **apps**.

An app is considered to be in the **foreground if any of the following is true:**

- It has a visible activity, whether the activity is started or paused.

- It has a foreground service.

- Another foreground app is connected to the app, either by binding to one of its services or by making use of one of its content providers. For example, the app is in the foreground if another app binds to its:

  - IME (Input Method Editor)

  - Wallpaper service

  - Notification listener

  - Voice or text service

If none of those conditions is true, the app is considered to **be in the background**.

Background Services

- While an app is in the foreground, it can create and **run both foreground and background services**

  - When an app goes into the background, it has a window of several minutes in which it is still allowed to create and use services

  - At the end of that window, the app is considered to be idle.

    - The system stops the app's background services, just as if the app had called the services' **Service.stopSelf()** methods.

- So any operation saving the state must be done in the **onDestroy** method (see later)

- To **reduce the danger** of apps sucking resources while in the background

  - Android 8.0 (API level 26) imposes limitations on what apps can do while running in the background

  - The system doesn't allow a background app to create a background service

  - Android 8.0 requires the apps to call **startForegroundService**() to start a new service in the foreground
    - After the system has created the service, the app has five seconds to call the service's startForeground() method to show the new service's user-visible notification.

  - If the app **does not call startForeground() within the time limit**, the system stops the service and declares the app to be ANR.

- A service must be declared in the **Manifest.xml** file
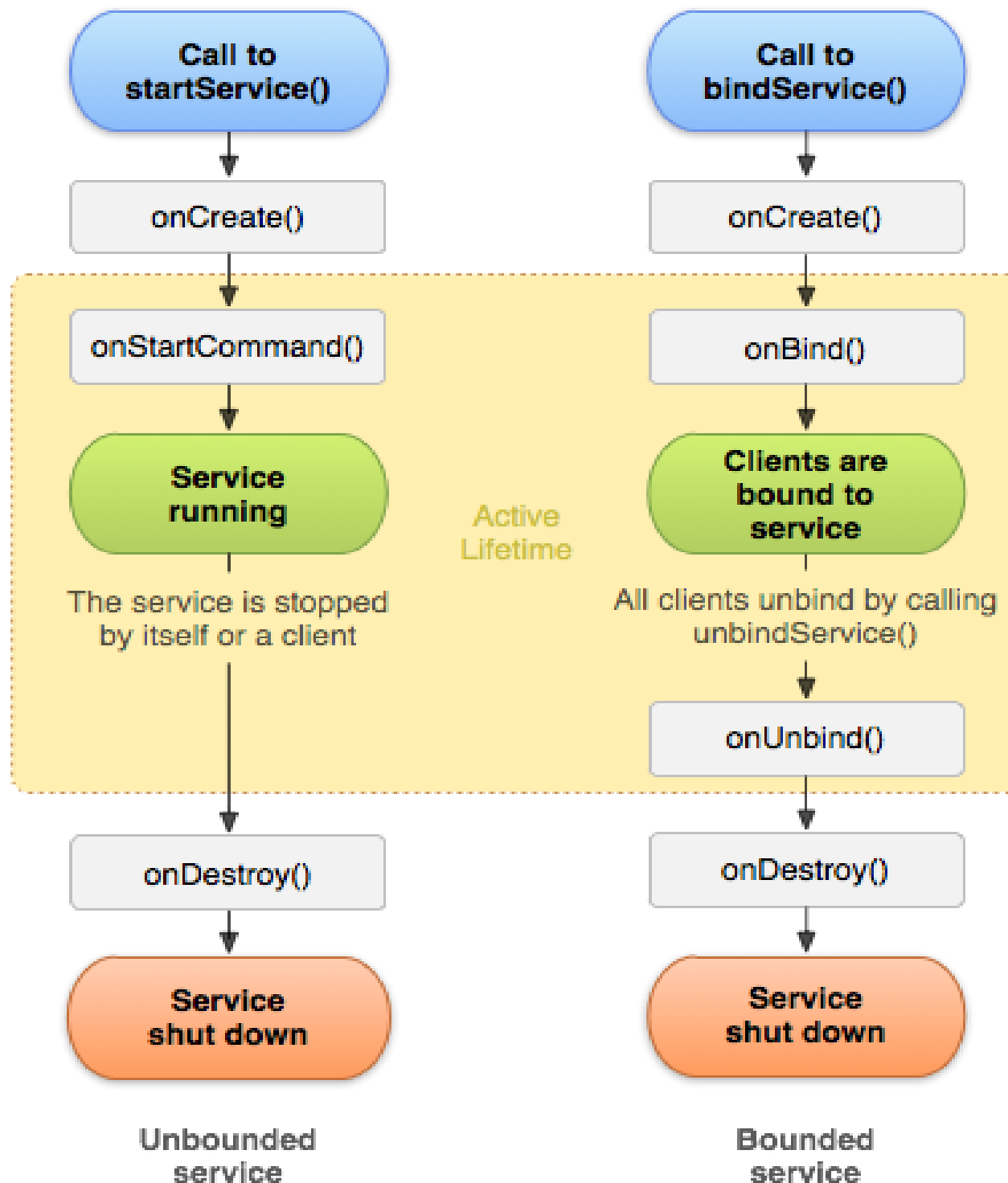
  - a common **error is to forget** and then wonder why the service does not start (no error returned)

```
<manifest ... >

    ...

    <application ... >

        <service android:name=".ExampleService" />

        ...

    </application>

</manifest>
```

You can ensure that your service is available to only your app by including the **android:exported** attribute and setting it to false.

Lifecycle of services

```
class ExampleService : Service() {
    private var startMode: Int = 0         // indicates how to behave i
    private var binder: IBinder? = null    // interface for clients tha
    private var allowRebind: Boolean = false   // indicates whether onRebin

    override fun onCreate() {
        // The service is being created
    }

    override fun onStartCommand(intent: Intent?, flags: Int, startId: Int):
        // The service is starting, due to a call to startService()
        return startMode
    }

    override fun onBind(intent: Intent): IBinder? {
        // A client is binding to the service with bindService()
        return binder
    }

    override fun onUnbind(intent: Intent): Boolean {
        // All clients have unbound with unbindService()
        return allowRebind
    }

    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
    }
}
```

https://developer.android.com/images/service_lifecycle.png

Main Methods

- **OnCreate()**: invokes this method to perform one-time setup procedures when the service is initially created.

- **OnStartCommand()**: invokes this method by calling *startService()* when another component (such as an activity) requests that the service be started.

- **OnBind()**: invokes this method by calling *bindService()* when another component wants to bind with the service.

- **OnDestroy()**: invokes this method when the service is no longer used and is being destroyed.

Starting a Service

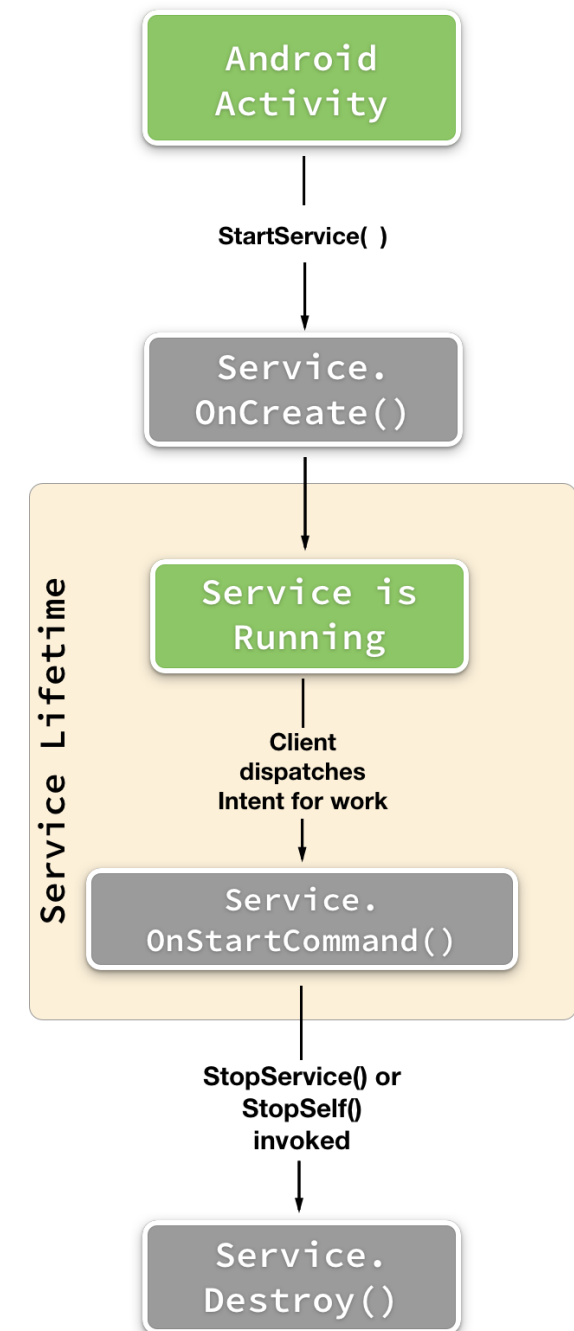- You can start a service from an activity or other application component by passing an Intent to **startService**() or **startForegroundService**()

- The Android system calls the service's **onStartCommand**() method and passes it the Intent, which specifies which service to start
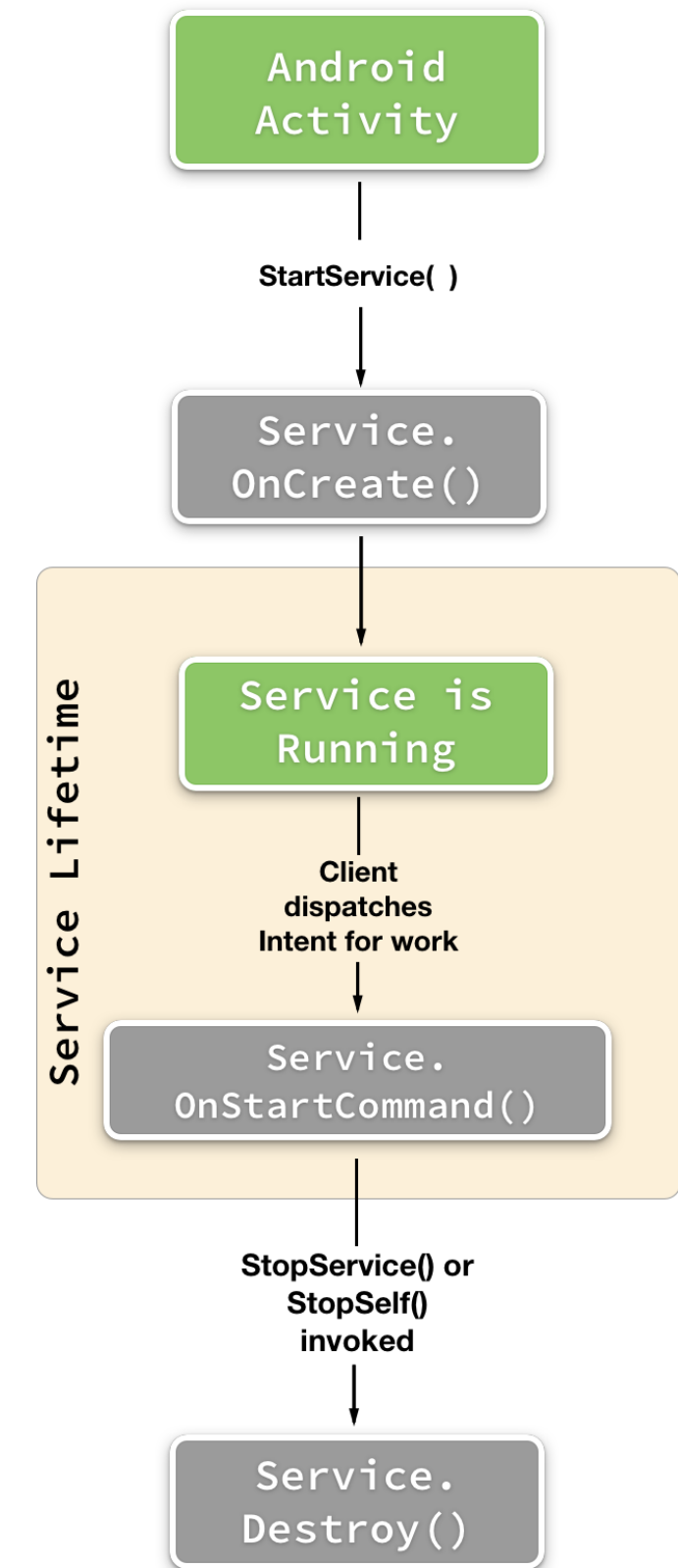
```
Intent(this, HelloService::class.java).also { intent ->
    startService(intent)
}
```

Lifecycle

- **A started service** is one that another components starts by calling ***startService()*** which results in a call to the service's ***OnStartCommand()***.

- When a service is started, it has a **lifecycle that is independent of the component** that started it.

  - The service can run in the background indefinitely, even if the component that started it is destroyed.

  - The service should stop itself when its job is completed by calling ***stopSelf()***.

  - Or another component can stop it by calling ***stopService()***.

Android
Activity

StartService( )

Service.
OnCreate()

Service is
Running

Client
dispatches
Intent for work

Service Lifetime

Service.
OnStartCommand()

StopService() or
StopSelf()
invoked

Service.
Destroy()

Location in the background

https://developer.android.com/training/location

Active Location Seeking

- You may want to seek locations actively
  - this costs a considerable **amount of battery**
  - do it only if needed
  - do not **start/stop unnecessarily location seeking**
    - once started let it run until needed
  - **stop it as soon as possible**

Request location permissions

- **Types of location access**

  - **Category**: either foreground location or background location

  - **Accuracy**: Either precise location or approximate location

```xml
<!-- Recommended for Android 9 (API level 28) and lower. -->
<!-- Required for Android 10 (API level 29) and higher. -->
<service
    android:name="MyNavigationService"
    android:foregroundServiceType="location" ... >
    <!-- Any inner elements would go here. -->
</service>
```

```xml
<manifest ... >
  <!-- Always include this permission -->
  <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />

  <!-- Include only if your app benefits from precise location access. -->
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
</manifest>
```

- In your activity **onCreate()** method, create an instance of the Fused Location Provider Client

```kotlin
private lateinit var fusedLocationClient: FusedLocationProviderClient

override fun onCreate(savedInstanceState: Bundle?) {
    // ...

    fusedLocationClient = LocationServices.getFusedLocationProviderClient(this)
}
```

- Get the last known location

```kotlin
fusedLocationClient.lastLocation
        .addOnSuccessListener { location : Location? ->
            // Got last known location. In some rare situations this can be null.
        }
```

Location Request

- Enables setting the location request parameters

```
fun createLocationRequest() {
    val locationRequest = LocationRequest.create()?.apply {
        interval = 10000
        fastestInterval = 5000
        priority = LocationRequest.PRIORITY_HIGH_ACCURACY
    }
}
```

- **Update interval**

  - **SetInterval ()** – This method sets the rate in milliseconds at which your app prefers to receive location updates.

- **Fastest update interval**

  - **SetFastestInterval ()** - This method sets the **fastest** rate in milliseconds at which your app can handle location updates. The Google Play services location APIs send out updates at the fastest rate that any app has requested with **SetInterval ()**.

- **Priority**

  - **SetPriority ()** - This method sets the priority of the request, which gives the Google Play services location services a strong hint about which location sources to use. The following values are supported:
    PRIORITY_BALANCED_POWER_ACCURACY -   PRIORITY_HIGH_ACCURACY •
    PRIORITY_LOW_POWER   PRIORITY_NO_POWER

Why a request?

- **Remember: your app never calls Android services**
  - It merely request them and Android can refuse them
  - This is why when you launch a **new Activity** you create an **Intent** that is satisfied by Android for reasons of force-majeure
    - In the case of location services your activity/service will request to use the location services
      - Always make sure to have a plan B if that request fails (rare) or is delayed (quite frequent)

Request location updates

- Before requesting location updates, your app must connect to location services and make a location request

- Once a location request is in place you can start the regular updates by calling **requestLocationUpdates()**

```kotlin
override fun onResume() {
    super.onResume()
    if (requestingLocationUpdates) startLocationUpdates()
}


private fun startLocationUpdates() {
    fusedLocationClient.requestLocationUpdates(locationRequest,
            locationCallback,
            Looper.getMainLooper())
}
```

Callback

- A callback is an object of type **LocationCallback** which must implement the method **onLocationResult**

```kotlin
private lateinit var locationCallback: LocationCallback

// ...

override fun onCreate(savedInstanceState: Bundle?) {
    // ...

    locationCallback = object : LocationCallback() {
        override fun onLocationResult(locationResult: LocationResult?) {
            locationResult ?: return
            for (location in locationResult.locations){
                // Update UI with location data
                // ...
            }
        }
    }
}
```

to get the location off the results
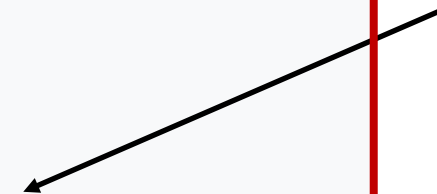
Stopping

- Consider whether you want to **stop the location updates** when the activity is no longer in focus, such as when the user switches to another app or to a different activity in the same app

- This can be handy to **reduce power consumption**, provided the app doesn't need to collect information even when it's running in the background

```kotlin
override fun onPause() {
    super.onPause()
    stopLocationUpdates()
}

private fun stopLocationUpdates() {
    fusedLocationClient.removeLocationUpdates(locationCallback)
}
```

Tracking locations in the background

- Tracking locations in the background requires an **Service**
  - create a Service (LocationIntent)
  - Add code in **onStartCommand()** method to capture the identification of the location
  - Create **startLocationUpdates()** method and Tasks
    - define what to do when the creation of the task is successful
    - define what to do when unsuccessful

- The parameters of requestLocationUpdates also change so to pass the intent rather than the location callback object (and the Looper)

in the onclicklistener for start

- StartLocationUpdates()

```kotlin
private fun startLocationUpdates() {
    Log.e( tag: "Location update",  msg: "Starting...")
    //  start receiving the location update

    val intent = Intent(ctx, LocationService::class.java)
    mLocationPendingIntent =
        PendingIntent.getService(ctx,
            requestCode: 1,
            intent,
            PendingIntent.FLAG_UPDATE_CURRENT
        )

    val locationTask = mFusedLocationClient.requestLocationUpdates(
        mLocationRequest,
        mLocationPendingIntent!!
    )
    locationTask.addOnFailureListener { e ->
        if (e is ApiException) {
            e.message?.let { Log.w( tag: "MapsActivity", it) }
        } else {
            Log.w( tag: "MapsActivity", e.message!!)
        }
    }
    locationTask.addOnCompleteListener { it: Task<Void!>
        Log.d(
            tag: "MapsActivity",
            msg: "starting gps successful!"
        )
    }

}
```

The Location Service

- Location service

```kotlin
class LocationService : Service {
    private var mCurrentLocation: Location? = null
    private var mLastUpdateTime: String? = null

    private var startMode: Int = 0              // indicates how to behave if the service is killed
    private var binder: IBinder? = null         // interface for clients that bind
    private var allowRebind: Boolean = false    // indicates whether onRebind should be used

    constructor(name: String?) : super() {}
    constructor() : super() {}

    override fun onCreate() {...}

    override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {...}

    override fun onBind(intent: Intent): IBinder? {
        // A client is binding to the service with bindService()
        return null
    }

    override fun onUnbind(intent: Intent): Boolean {
        // All clients have unbound with unbindService()
        return allowRebind
    }

    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
        Log.e( tag: "Service",  msg: "end")
    }
}
```

```kotlin
override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
    // The service is starting
    if (LocationResult.hasResult(intent!!)) {
        val locResults = LocationResult.extractResult(intent)
        for (location in locResults.locations) {
            if (location == null) continue
            //do something with the location
            Log.i( tag: "This is in service, New Location",  msg: "Current location: $location")
            mCurrentLocation = location
            mLastUpdateTime = DateFormat.getTimeInstance().format(Date())
            Log.i( tag: "This is in service, MAP",  msg: "new location " + mCurrentLocation.toString())
            // check if the activity has not been closed in the meantime
            if (MapsActivity.getActivity() != null) // any modification of the user interface must be done on the UI Thread. The Intent Service is running
            // in its own thread, so it cannot communicate with the UI.
                MapsActivity.getActivity()?.runOnUiThread(Runnable {
                    try {
                        MapsActivity.getMap().addMarker(
                            MarkerOptions().position(
                                LatLng(
                                    mCurrentLocation!!.latitude,
                                    mCurrentLocation!!.longitude
                                )
                            )
                                .title(mLastUpdateTime)
                        )
                        val zoom = CameraUpdateFactory.zoomTo( zoom: 15f)
                        // it centres the camera around the new location
                        MapsActivity.getMap().moveCamera(
                            CameraUpdateFactory.newLatLng(
                                LatLng(
                                    mCurrentLocation!!.latitude,
                                    mCurrentLocation!!.longitude
                                )
                            )
                        )
                        // it moves the camera to the selected zoom
                        MapsActivity.getMap().animateCamera(zoom)
                    } catch (e: Exception) {
                        Log.e( tag: "LocationService",  msg: "Error cannot write on map " + e.message)
                    }
                })
```

Note: it may return a number of locations (some of them may be null!!)

Never Ending Processes

Working in the background

- You use a background or foreground service to provide **continuous data collection or processing**

  - while the app is no longer in the foreground (i.e. it has been minimised and another app is displayed)

  - or even when the app has been closed by the user -

  - or killed by Android (this happens when the operating system is running out of resources, e.g. memory).

- A typical example of an app that requires a never ending background service is a **pedometer that tracks your steps 24/7**.

- No matter if the app is not in the foreground, you still want to count your steps.

© Po Yang, University of Sheffield

What do you need?

- You just need a simple trick

- Create a **delayed broadcast** to a receiver in the **onDestroy** of the service

  - The receiver will restart the service

- This tricks work very reliably

  - Tested on our OAK app **Active10** which has over 870,000 downloads and works on nearly 10,000 devices

```java
public class SensorRestarterBroadcastReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        Log.i(SensorRestarterBroadcastReceiver.class.getSimpleName(),
                    "Service Stops! Oooooooooooopppppsssss!!!!");
        context.startService(new Intent(context, SensorService.class));;
    }

}
```

```xml
<application
…
<receiver
    android:name=".SensorRestarterBroadcastReceiver"
    android:enabled="true"
    android:exported="true"
    android:label="RestartServiceWhenStopped">
</receiver>
…
</application>
```

© Po Yang, University of Sheffield

```java
@Override
public void onDestroy() {
    super.onDestroy();
    Log.i("EXIT", "onDestroy!");
    Intent broadcastIntent = new Intent(this, SensorRestarterBroadcastReceiver.class);
    sendBroadcast(broadcastIntent);

}
```

You may wan to delay the sending by a couple of seconds so to wait for the process to die

```java
new java.util.Timer().schedule(

        new java.util.TimerTask() {

            @Override

            public void run() {

                // your code here

            }

        },

        2000

);
```

- This method will not work in Android 7 onwards

  - that is because processes launched with broadcast receivers are now killed or severely downgraded after the receiver code is no longer running

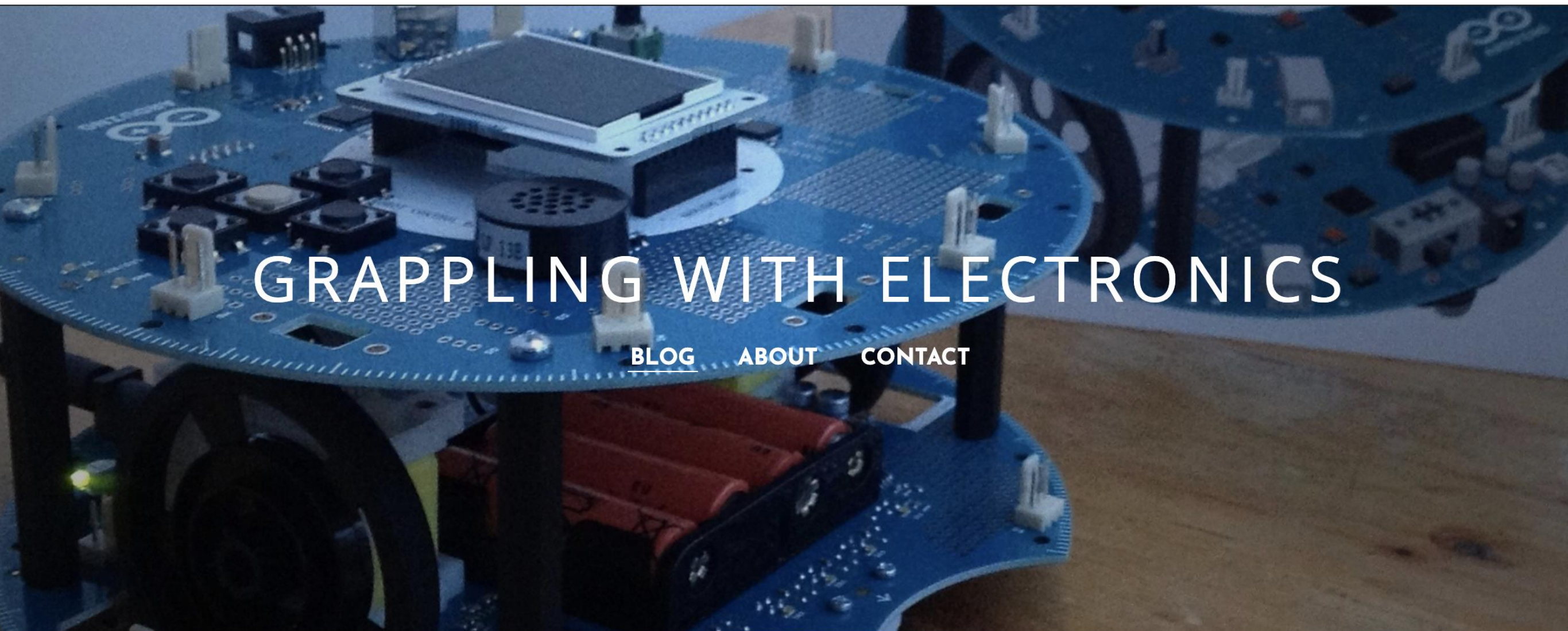- You will need to use a **JobService**

Job Services

- A **specialised type of services** that performs based on conditions (e.g. when the phone is charging)

- it must be declared in the manifest as any other service

- it must be set up with conditions of execution attached

Full explanation with code

GRAPPLING WITH ELECTRONICS

BLOG   ABOUT   CONTACT

Creating a never ending background service in Android > 7
17/11/2019

0 Comments

In a previous post, I explained how to create a never ending background service. That solution worked for Android up to Version 7. After that, that method has started misbehaving on an increasing number of vendor specific Android implementations (e.g. Xiaomi) and then finally most of the vendors.
So it is time to give an update.

Author

I am professor of Computer Science at The University of Sheffield.
My research is in large scale data acquisition and analysis with a particular focus on

46

Doze and App Standby

aggressive battery saving strategies

https://developer.android.com/training/monitoring-device-state/doze-standby.html
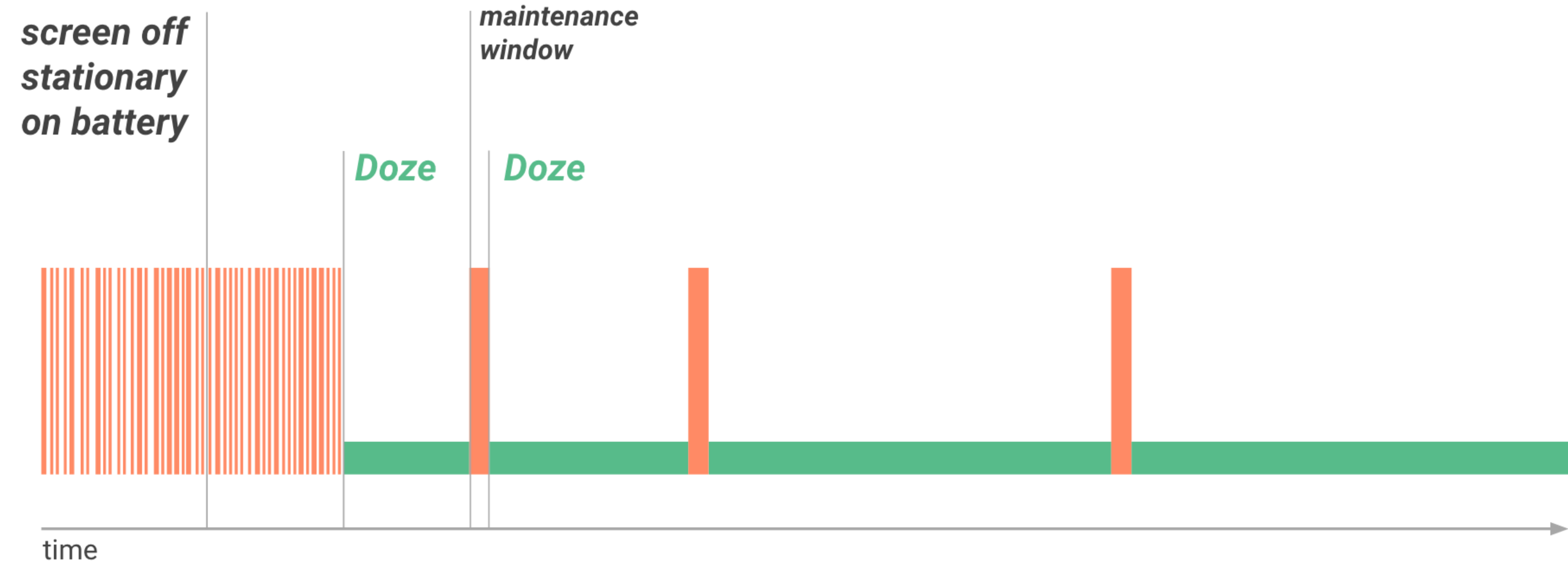
Doze

- If the device is left unplugged and stationary for a period of time, with the screen off, the device enters **Doze mode**

- In Doze mode, the system attempts to **conserve battery by restricting apps' access to network and CPU-intensive services**

    - It also prevents apps from accessing the network and defers their jobs, syncs, and standard alarms.

- Periodically, the system exits Doze for a brief time to let apps complete their deferred activities. During this maintenance window, the system runs all pending syncs, jobs, and alarms, and lets apps access the network

Doze Restriction

- Network access is suspended.

- The system ignores wake locks.

- Standard AlarmManager alarms (including setExact() and setWindow()) are deferred to the next maintenance window.

  - Alarms set with setAlarmClock() continue to fire normally — the system exits Doze shortly before those alarms fire.

- The system does not perform Wi-Fi scans.

- The system does not allow sync adapters to run

This behaviour is being changed with new versions of Android

App Idle

- App Standby allows the system to determine that an app is idle when the user is not actively using it.

- Unless:

  - The user explicitly launches the app.

  - The app has a process currently in **the foreground** (either as an activity or foreground service, or in use by another activity or foreground service).

    Note: You should only use a foreground service for tasks the user expects the system to execute immediately or without interruption. Such cases include uploading a photo to social media, or playing music even while the music-player app is not in the foreground. You should not start a foreground service simply to prevent the system from determining that your app is idle.

Summary

- Part 1:
  - Services
  - IntentServices
- Part 2:
  - Location in the background
  - Tracking locations in the background
- Lab tutorial :
  - Maps, Locations and Services