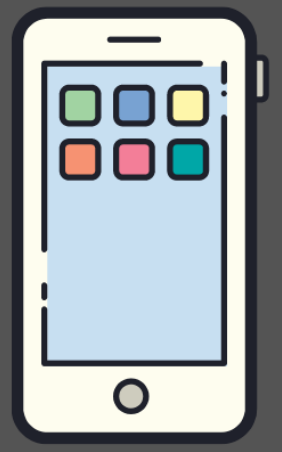


MOBILE DEVELOPMENT

Process Management

Services and General Background Processes

Once again… Disclaimer

- There's a lot of information to digest when it comes to process management in Android
  - Commonly used components being depreciated…
  - New components / architectures being introduced…
- So… discussion is going to be somewhat general
  - If you need to learn more, you'll have an idea of concepts that should be kept in mind as you wade through

Also, for the Kotlin peeps, you'd best want to look at **Coroutines** to construct asynch tasks

In the last part…

- We had a brief overview of concurrency and talked about threads in Android
- We should know that…
  - Any long running operation should be offloaded from the main thread so that onDraw() is not blocked
  - No UI update should be done outside of the main thread
    - If we need to update UI, we can utilize runOnUiThread() or a Handler linked to the main thread

Before we get into today's lesson…

AsynchTask (depreciated in API 30)

- Despite recently being depreciated, AsynchTask was a common method create background threads
  - You'll most likely run into it if you're looking up information
- Consists of 4 main parts:
  - `onPreExecute()` – called on UI thread pre-execution
  - `doInBackground(Params…)` – tasks performed asynchronously
  - `onProgressUpdate()` – can be indirectly called in the UI thread
  - `onPostExecute(Result)` – Process tasks after `doInBackground()`

Even if AsynchTasks aren't encouraged any more, the 4 main parts define appropriate concepts when dealing with processes

Outline

- Services
  - Bound vs Started Service
  - Lifecycle
  - Communication (BroadcastReceiver)
  - Design considerations
- General Background Processes

RECALL: Thread are killed when app is killed

- Threads are alright if you have simple tasks to do in app, but you wouldn't want to use these for:
  - Operations outside of the app
  - Periodic activities
  - Activities that require specific settings
    - E.g. run only when connected to a network
- Android offers more powerful components…

Services

- Are similar to Activities, but without a UI component
  - Runs in the *background*
  - Must be declared in the Manifest!
- Can be launched from activities
- Can continue to execute tasks after closing of is starting activity

exported:"false" just means the service is only available to the app which this is declared in

Services must be declared in the Manifest!

```
<service
    android:name=".MyService"
    android:exported="false"></service>
```

Services

- Started Services (unbound) are started via `startService()`
- Bound Services are bound / connected to a client Activity

- A Service can be either or even both at the same time

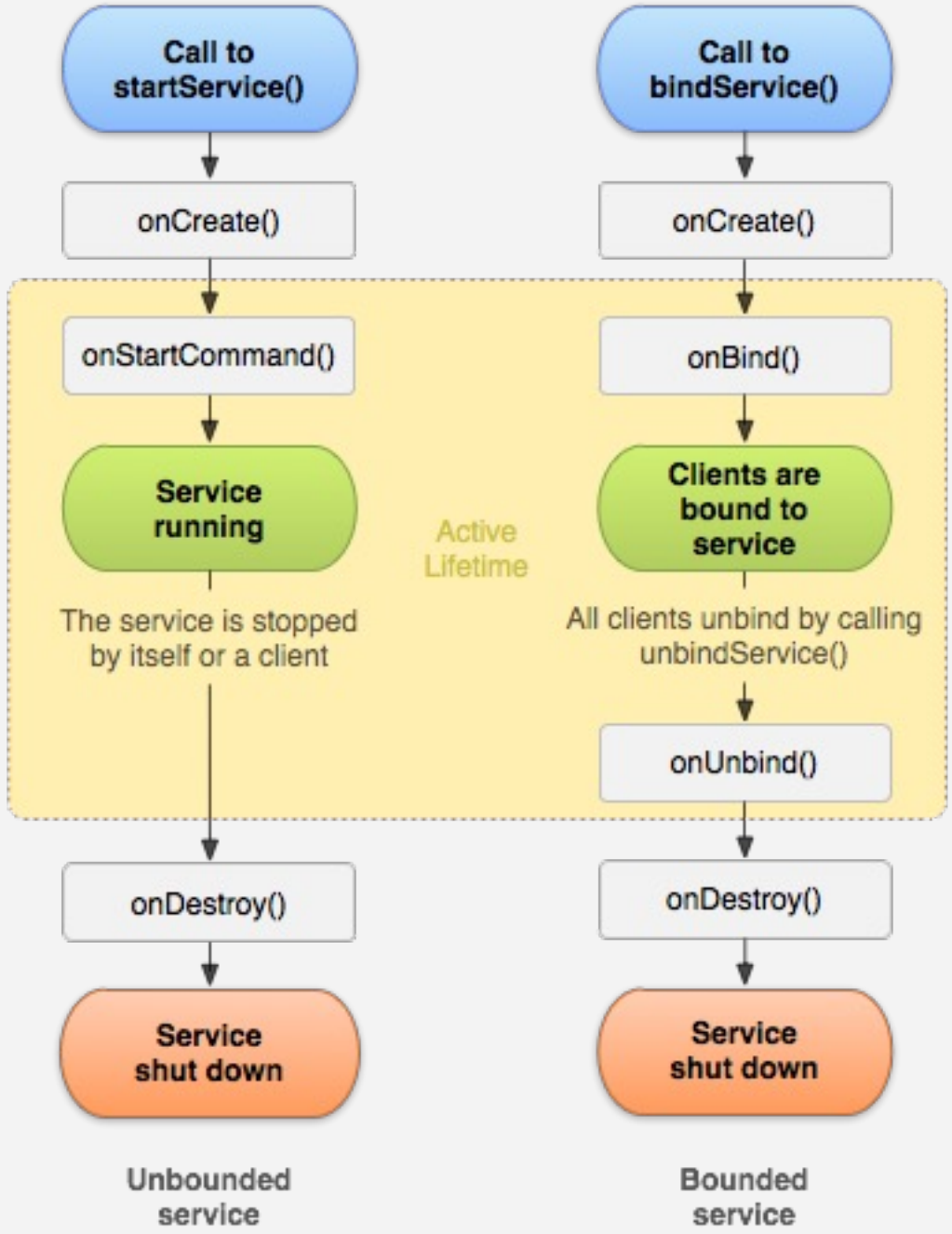


Image source: https://developer.android.com/guide/components/services

Services

- Has their own lifecycle… err… life
  - Continues to run until it is stopped (either by itself, what it is bound to, or by the OS)
  - Services have higher priority than activities

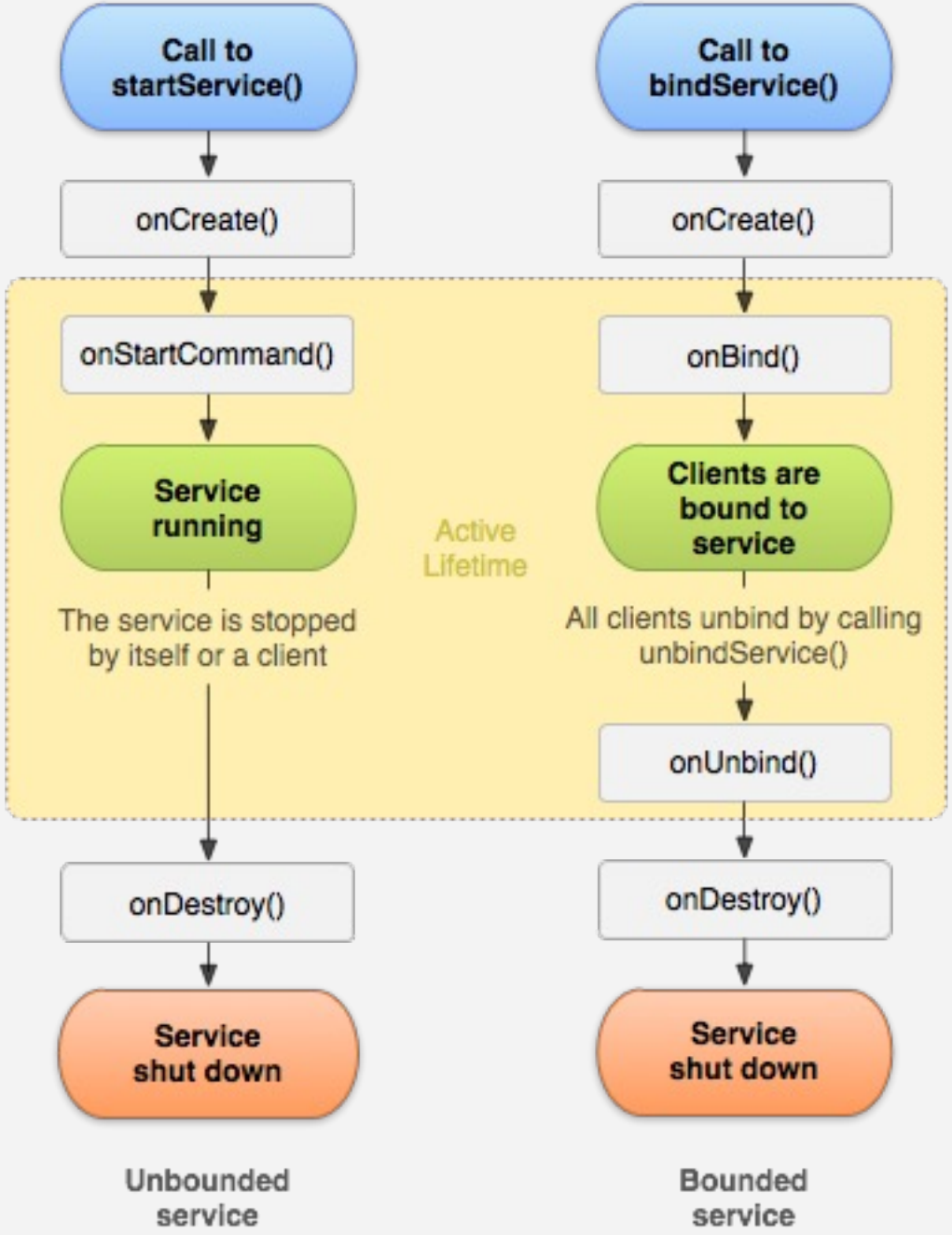


Image source: https://developer.android.com/guide/components/services

Services

- `OnCreate()`
  - Nothing much happens here since there aren't UI elements to setup

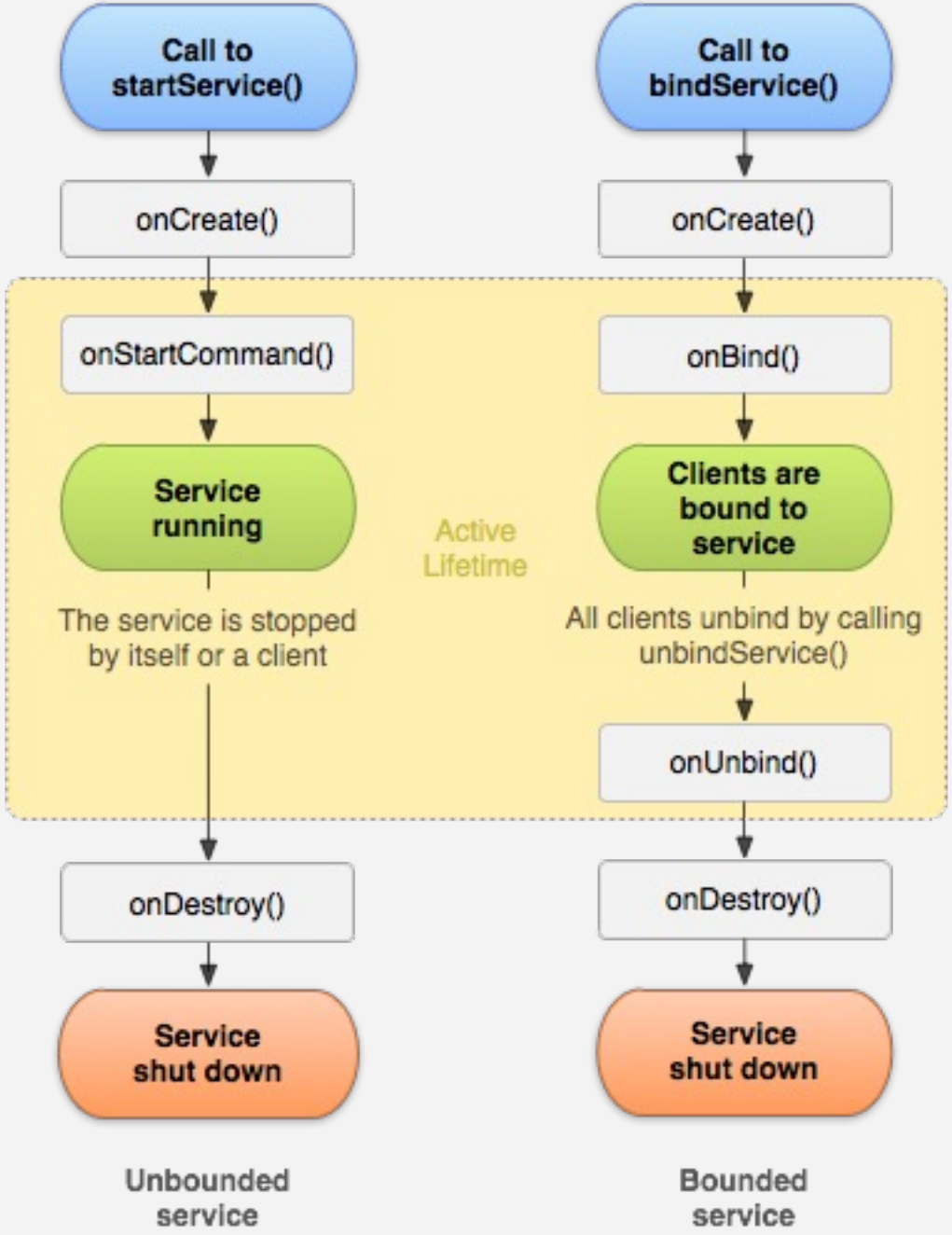


Image source: https://developer.android.com/guide/components/services

Services

- `onStartCommand()`
  - This method is called when the Service is sent a command by another component
  - Is not only called once, but whenever the Service receives an intent

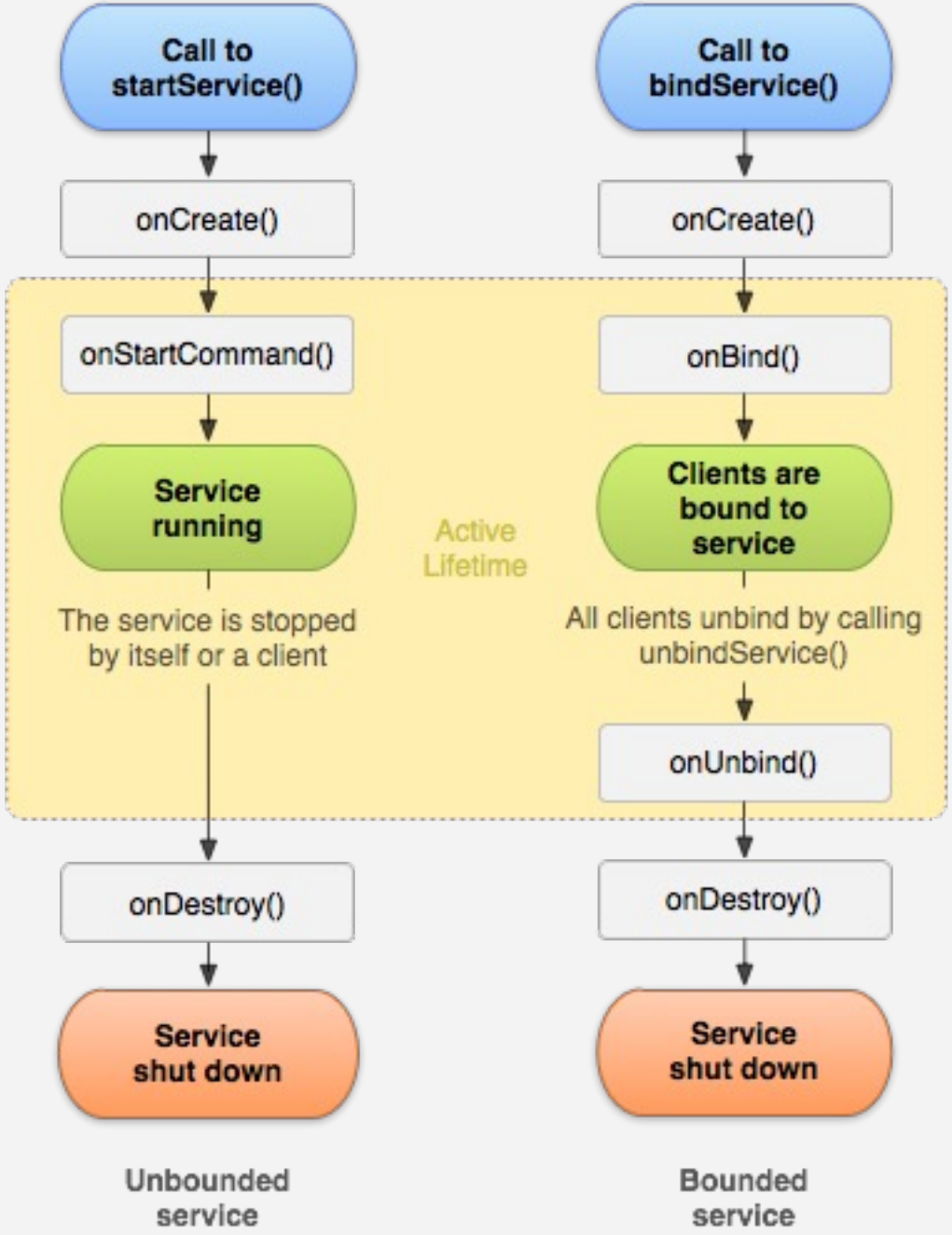


Image source: https://developer.android.com/guide/components/services

Services

- **onBind() & onUnbind()**
  - onBind() returns an IBinder object, which enables an Activity / Client direct access the Service
  - onUnbind() is for cleaning up purposes

Check out the 07C_General_Service Android Project for more info on Bound Services and a sample implementation ☺

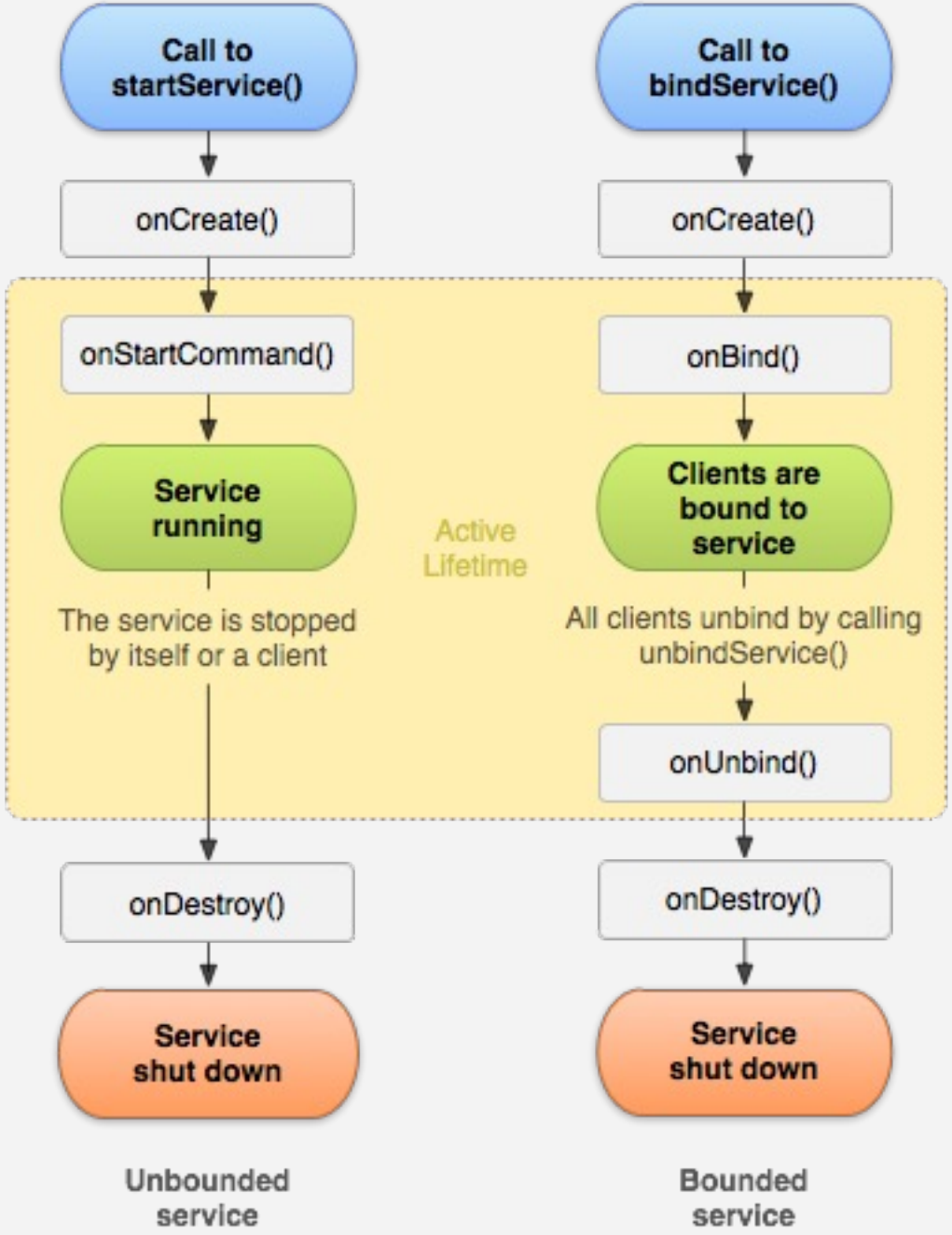


Image source: https://developer.android.com/guide/components/services

Services

- `onDestroy()`
  - Services need to manually be told to stop
  - From another component:
    - `stopService(Intent)`
  - Service can also call `stopSelf()`

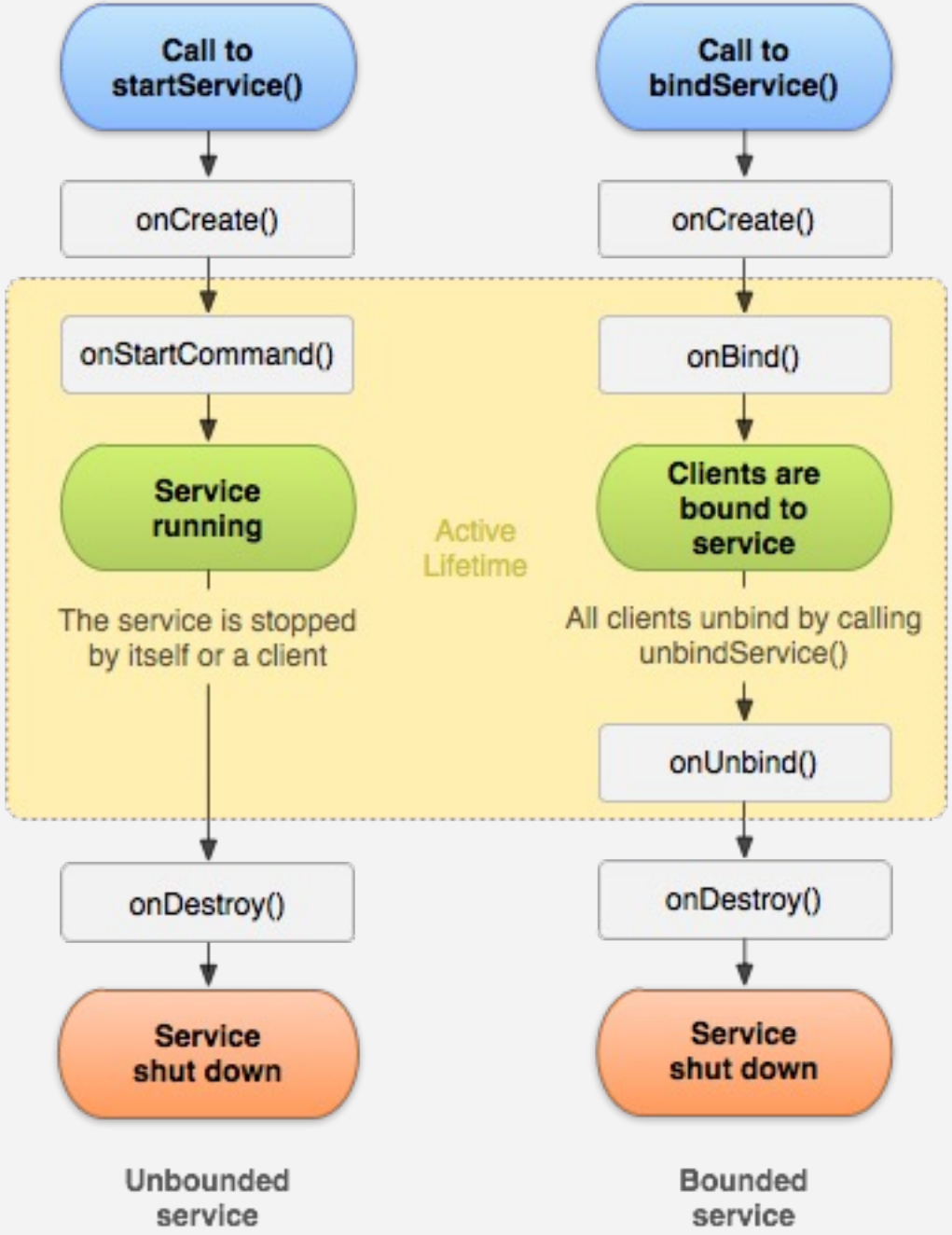


Image source: https://developer.android.com/guide/components/services

Services

- Are not separate processes
  - Runs within the process of the application
- Are not threads
  - Generic services run on the main thread by default
    - For other types, you might want to check what Looper they're on
  - However, we'd typically want to run tasks on a separate thread
  - "A *Service runs in the background*" doesn't mean its running on the background thread, but that it isn't seen by the user

Design Considerations with Services

- Start vs bind?
  - Start a service if you imagine it running independently from any front facing components of the activity (e.g. Activity)
    - E.g. Playing background music in a game app
  - Bind if the binding components requires interactions with the service's instance
    - E.g. Playing music with user controls
  - However, as mentioned before, you can also perform both

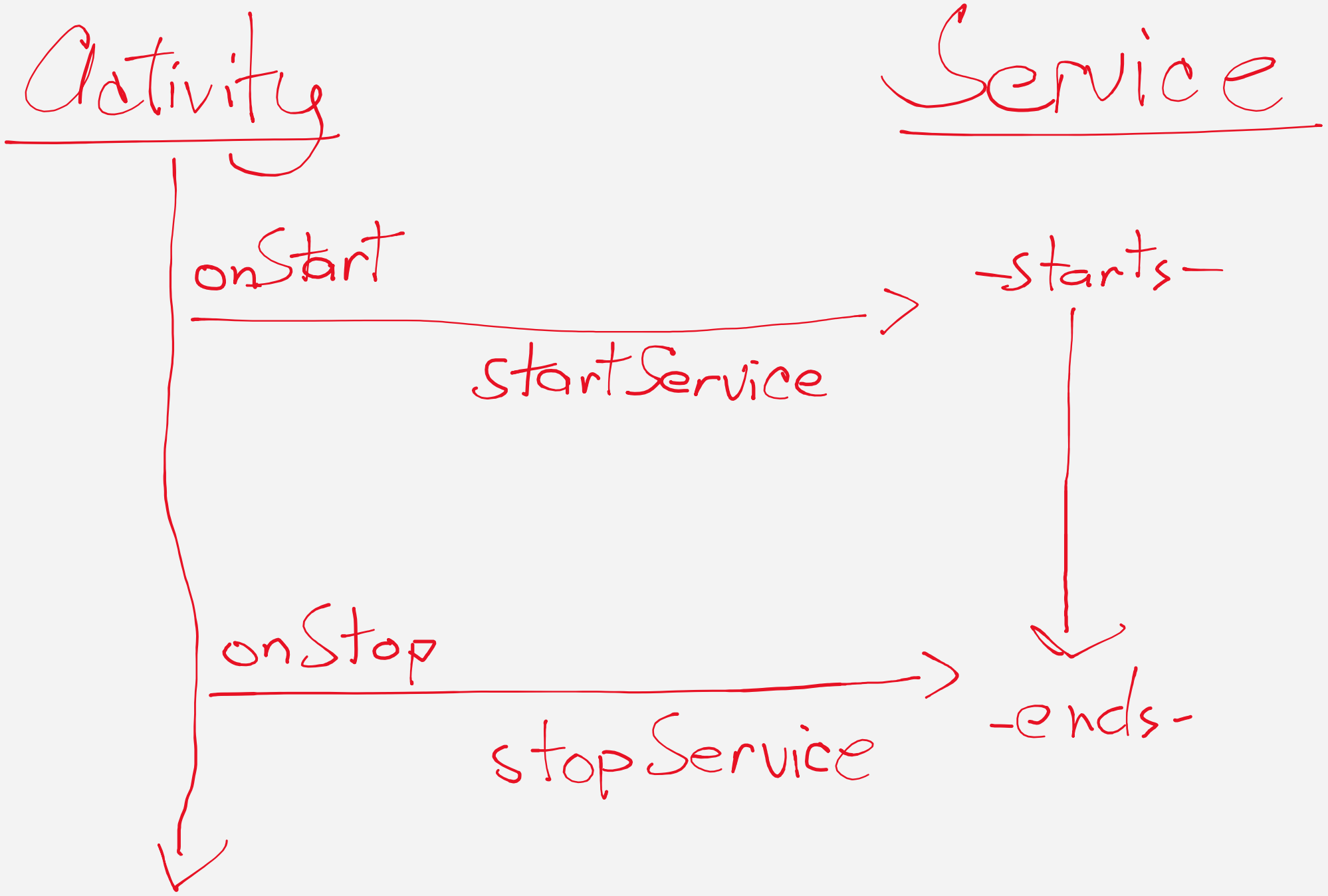
Design Considerations with Services

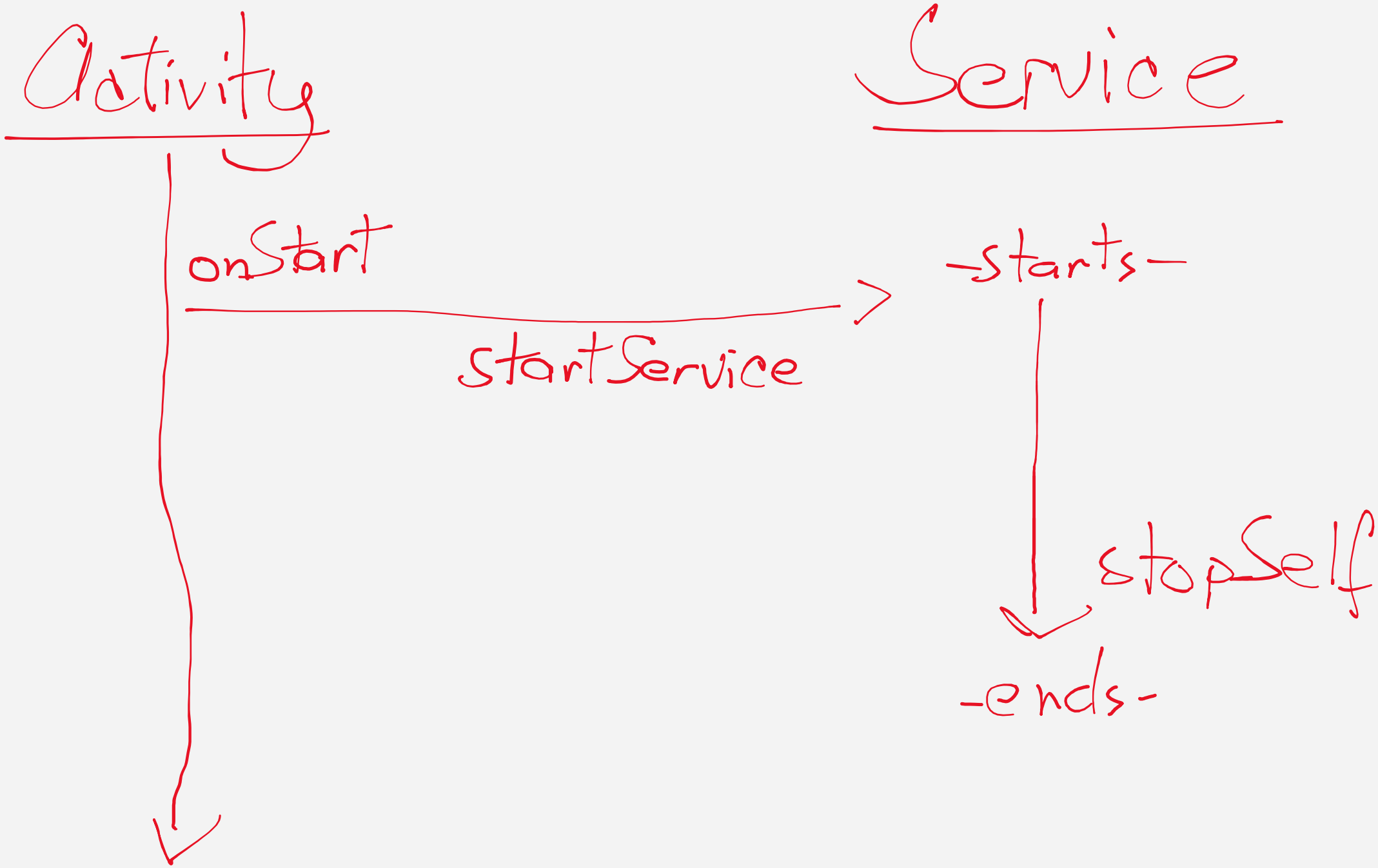
- Cleaning up
  - Like in the discussion with general threads, services (and any other background task) need to be planned out thoroughly – from start to end of their execution
  - If not structured properly, services will continue to execute after the launching / binding activity has finished
    - This can lead to higher battery consumption… and effect the phone's battery life

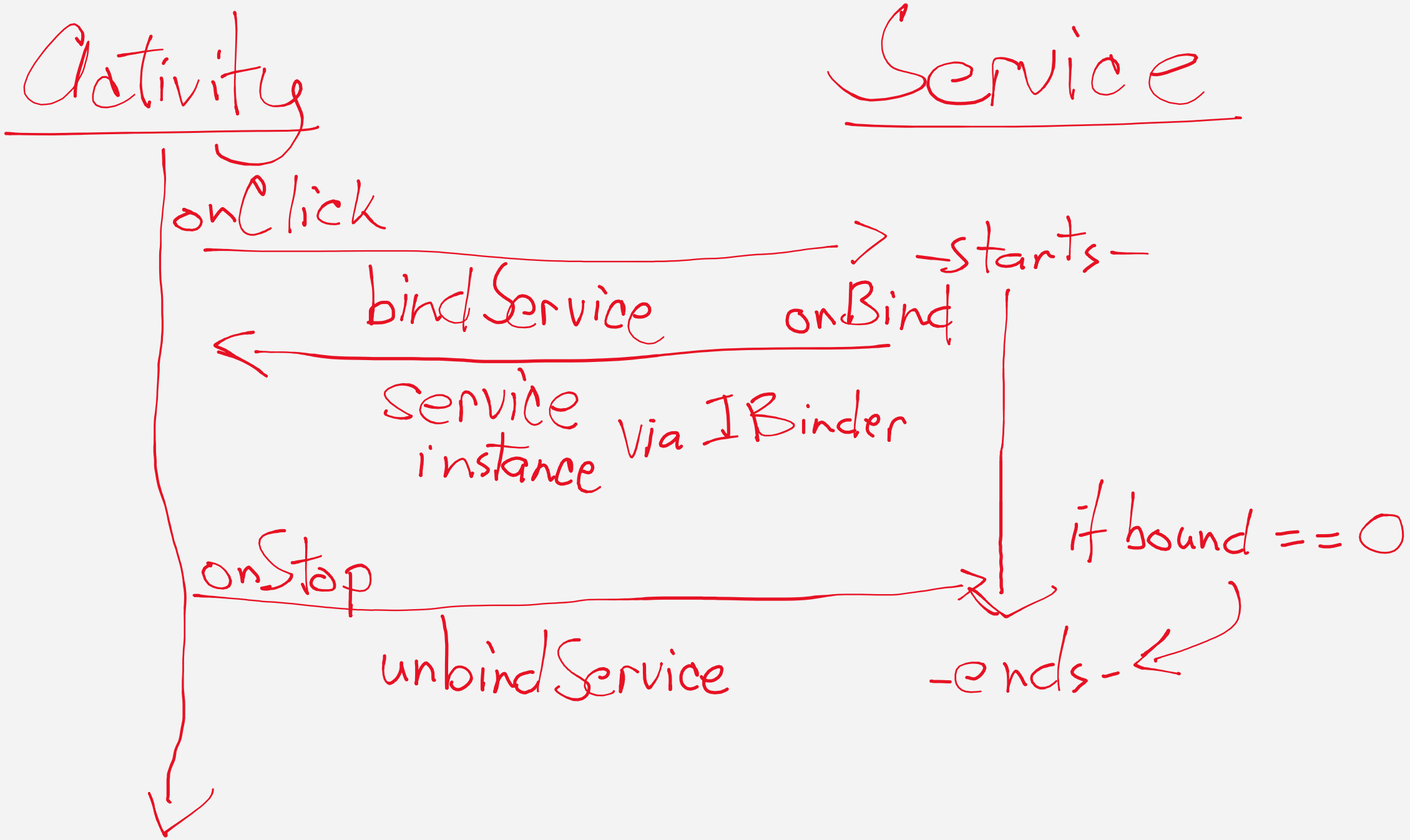
Design Considerations with Services

- Cleaning up
  - There's no one shot guide to cleaning up services as use cases vary according to app requirements
  - Be aware of when you need your service…
    - To start and/or be bound
    - To end and/or be unbound
  - Remember that the launching component could end at any time, so a service should be designed to

Note: Services can call stop themselves by calling this.stopSelf() within the service


Activity
Service
onStart
startService
-starts-
onStop
stopService
-ends-


Activity
Service
onStart
startService
-starts-
stopSelf
-ends-


Activity
Service
onClick
-starts-
bindService
onBind
service
instance
via IBinder
onStop
if bound == 0
unbindService
-ends-

Communicating with a Service

- Sometimes, you might need your Service and Activity to interact with each other
- Some manners to do so would be through:
  - Intent data (the intent used when starting the service)
  - Handler (via setter method)
  - Service Binding (calling methods of service)
  - Broadcaster + Receiver

Broadcasts and Receivers

- Not solely related to processes
- Broadcasts can be used in order to send messages across the Android System
  - We broadcast an Intent with an Action and Data
  - `sendBroadcast()`

```
Intent i = new Intent();
i.setAction("declare your custom action tag here");
i.putExtra(SOME_KEY, some_data);
sendBroadcast(i);
```

Broadcasts and Receivers

- Not solely related to processes
- Receivers can be used to receive / listen for broadcasts
  - We would use a BroadcastReceiver
    - Define an IntentFilter or a list of intents to listen for
    - Utilize `onReceive(Context, Intent)` method to handle incoming broadcasts

BroadcastReceiver

One can register the BroadcastReceiver in code as such…

In the MainActivity

```
private BroadcastReceiver receiver = new MyReceiver();
private boolean registered = false;

IntentFilter filter = new IntentFilter();
filter.addAction("ph.edu.dlsu.ccs.a08a_jobsandbroadcasts.MESSAGE_JOB_FIBBONACI");
if(!registered){
    registerReceiver(receiver, filter);
    registered = true;
}
.
.
public class MyReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Log.d("JOB_SCHEDULER","MyReceiver onReceive");
        if(intent.getAction().equals("ph.edu.dlsu.ccs.a08a_jobsandbroadcasts.MESSAGE_JOB_FIBBONACI")) {
            String fib = intent.getStringExtra("FIBBONACI");
            ansFld.setText(fib);
        }
    }
}
```

BroadcastReceiver

- Alternatively, one can register BroadcastReceivers in the Manifest

```
<receiver android:name=".MyReceiver">
    <intent-filter>
        <action android:name="android.intent.action.ACTION_POWER_CONNECTED"/>
        <action android:name="android.intent.action.ACTION_POWER_DISCONNECTED"/>
        <action android:name="android.intent.action.BATTERY_CHANGED"/>
        <action android:name="android.intent.action.BATTERY_OKAY"/>
    </intent-filter>
</receiver>
```

BroadcastReceiver

- In some cases, you might want to listen for common Intents send throughout the systems

- Check out…

  https://www.tutorialspoint.com/android/android_intent_standard_actions.htm

Broadcast + Receiver + Service

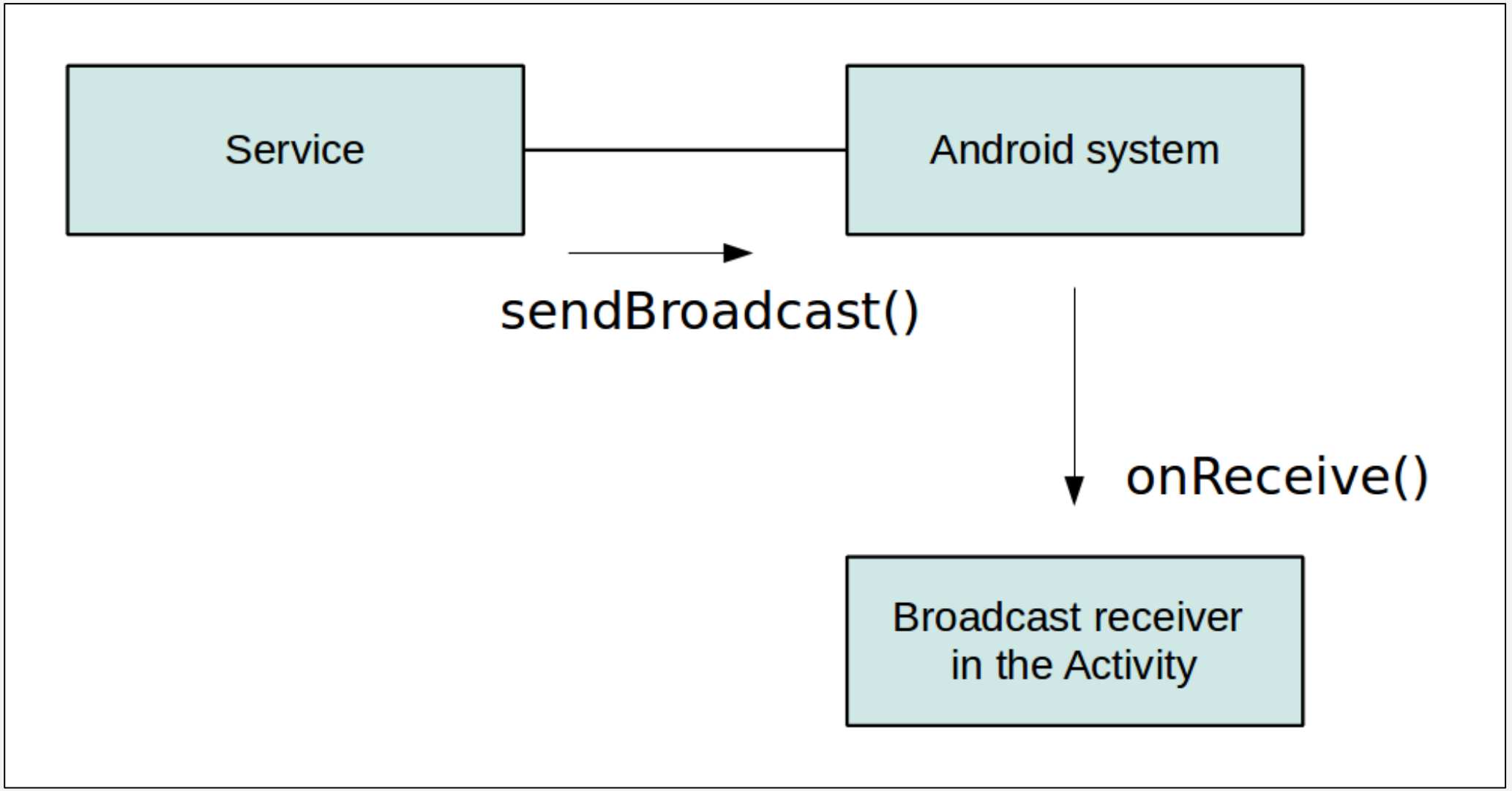


Image source: https://www.vogella.com/tutorials/AndroidServices/article.html

Any questions so far?

Background Processing

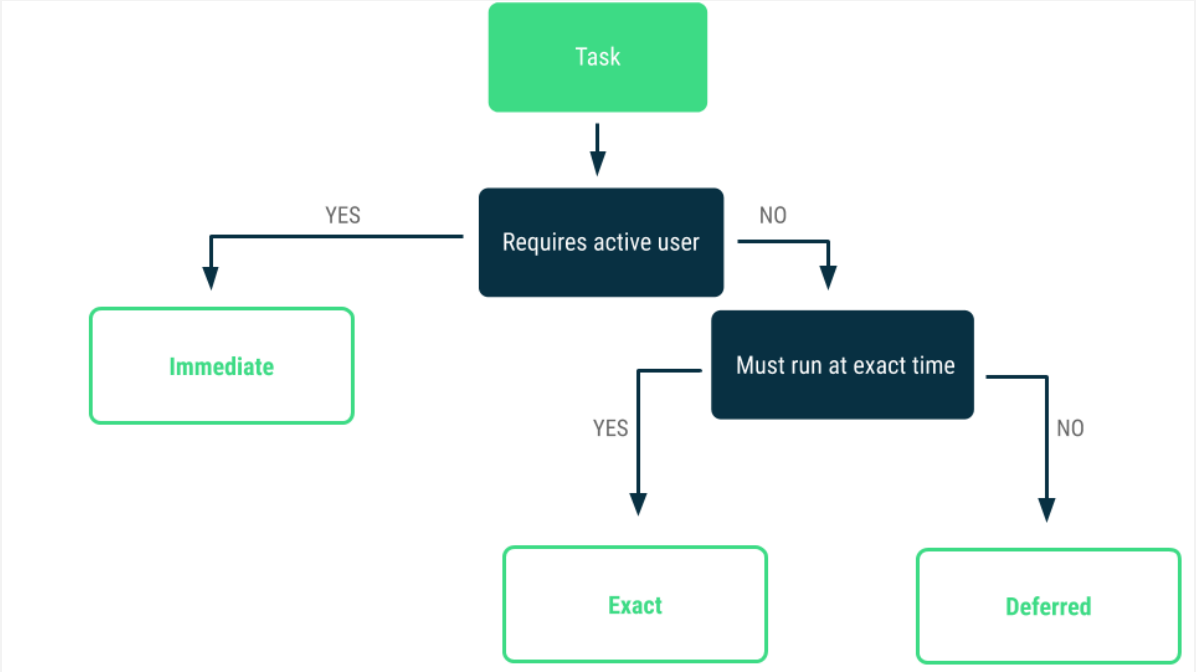
Background Processing

- So far we've discussed Threads and Services as means to offload work; however, they're two different things
  - Threads are for offloading work from the main thread
    - Best for when a task has some interaction with the user
    - Tied to the app's lifecycle
  - Services are meant for work to be done outside of user-facing components
    - Best when a task has no interaction with the user
    - Can have a separate "life" from the running app

Threads are often used in coordination with other background API

Background Processing

- Immediate
  - Threads, Coroutines
  - Continuing: WorkManager,
  - User-facing: Foreground service
- Deferred (to any time in the future)
  - WorkManager
- Exact (point in time)
  - AlarmManager



These are what Android Documentations recommends. We don't have to adhere to them exactly, but it'll be good to know what's options we have in creating out solutions.

Image sauce: https://developer.android.com/guide/background

For executing threads…

TimerTask

- Can schedule a task for one-time or repeated execution using a Timer object
- Very simple and straightforward way to move off the main thread
- Clean up:
  - cancel() – terminates timer
  - purge() – cancels queued tasks

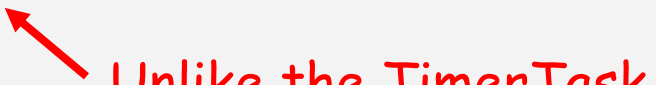
```
Timer t = new Timer();
tTask = new TimerTask() {
        public void run() {
                // action here
        }
};

// runs a single task with a 1 second delay
t.schedule(tTask, 1000);

// runs a task after 1 second delay every 5 seconds
t.schedule(tTask, 1000, 5000);
```

ExecutorService

- Provides a means to manage tasks off the main thread
- Can handle single or multiple threads
  - Executors.newSingleThreadExecutor();
  - Executors.newFixedThreadPool(int threadPoolSize);
- Clean up:
  - .shutdown() – queued tasks finish executed; no new tasks accepted
  - .shotdownNow() – all tasks are immediately interrupted



ExecutorServices are preferred over Timers because of their flexibility. However, if you're looking for something quick and simple to build, Timers are more straight forward.

ScheduledExecutorService

- Is an ExecutorService that can run with a delay or periodically
  - Single run w/ delay
    - schedule (runnable, delay, TimeUnit)
  - Periodic run
    - scheduleAtFixedRate(runnable, initialDelay, period, TimeUnit)
    - scheduleWithFixedDelay(runnable, initialDelay, delay, TimeUnit)

For deferring tasks to another time…

And for tasks that are mostly non-user-facing

AlarmManager

- Gives you a way to perform time-based operations outside the lifetime of your application.
- Can be used to initiate a long-running operation, such as starting a service once a day to download a weather forecast
- Schedule ahead of time too reduce the need for a constant running background processes

Note: Starting Oreo (Android 8), if you want to start a service, it would be best to start it in the foreground using startForgroundService() in the calling component and startForground() in the service component.

AlarmManager

Note: After API 19, alarm delivery isn't exact to minimize wake up and battery usage. You can use setExact() or setExactAndAllowWhileIdle(), but please consult the documentation regarding what you need.

```
Intent alarmIntent = new Intent(ACTION_FROM_ALARM);
alarmCount++;
alarmIntent.putExtra("ID", alarmCount);

PendingIntent pendingIntent = PendingIntent.getBroadcast(
        MainActivity.this.getApplicationContext(),
        1000000 + alarmCount,
        alarmIntent,
        0
);
```

IDs need to be unique or else they're overwrite the previous alarm set. You can also cancel the alarm with the ID. This can also be the current time so its unique.

The last parameter is a flag. Include FLAG_ONE_SHOT, FLAG_NO_CREATE, FLAG_CANCEL_CURRENT, FLAG_UPDATE_CURRENT, FLAG_IMMUTABLE. Usually set to 0.

```
AlarmManager manager = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
manager.set(AlarmManager.RTC_WAKEUP, 1000, pendingIntent);
```

Delay in milliseconds

RTC_WAKEUP has an alarm time based on the current system time and will wake up the device. If no wake up is needed, you can use RTC. See documentation for more info.

Can also be:

- setRepeating()
- setExact()
- And others found in the documentation

JobScheduler

- Allows for specific used-defined jobs to be executed
- Unlike the AlarmManager, you can batch jobs or even define scheduling criteria (e.g. job will execute only when phone is <criteria>)
- Requires:
  - JobInfo -> scheduling criteria
  - JobService -> implementing component of the job

JobService of JobScheduler

```
public class MyJobService extends JobService {

    @Override
    public void onCreate() {
        super.onCreate();
    }

    @Override
    public boolean onStartJob(JobParameters jobParameters) {
        // Your logic here
        return true;
    }

    @Override
    public boolean onStopJob(JobParameters jobParameters) {
        return false;
    }
}
```

Note: The JobService runs on a Handler to the Main Thread, so you'd best want to add in offloading capabilities when executing logic (e.g. threads).

Returning **true** implies that the job needs to continue running and will remain active until jobFinished() is called. Job's may be halted due to the scheduling criteria (e.g. loss of WiFi, low battery, etc.).
Returning **false** means the job has been finished and onStopJob won't be called.

Returning **true** implies that the job will be rescheduled. You'll have to handle the rescheduling however.
Returning **false** implies that the job has ended entirely.

JobScheduler

This is our custom JobService

```
ComponentName serviceComponent = new ComponentName(getApplicationContext(), MyJobService.class);

PersistableBundle bundle = new PersistableBundle();
bundle.putInt("ID", jobCount);

JobInfo.Builder builder = new JobInfo.Builder(jobCount, serviceComponent);
builder.setExtras(bundle);
builder.setMinimumLatency(1000);
builder.setOverrideDeadline(1000);
builder.setRequiresCharging(true);
builder.setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY);

JobScheduler jobScheduler = (JobScheduler) (getApplicationContext().getSystemService(JOB_SCHEDULER_SERVICE));
jobScheduler.schedule(builder.build());
```

A PersistableBundle is like any Bundle object we've used before (i.e. Intents), but this persists (i.e. can be stored in disk). Not a requirement.

Job ID. Like with the AlarmManager, this should be unique unless you want to modify the previous set job.

These aren't requirements, but can specify conditions for when you want the job to execute

WorkManager

- Android's workaround to different issues with implementing long running background tasks
- Factors in API and the phone's current status (focused on the battery)

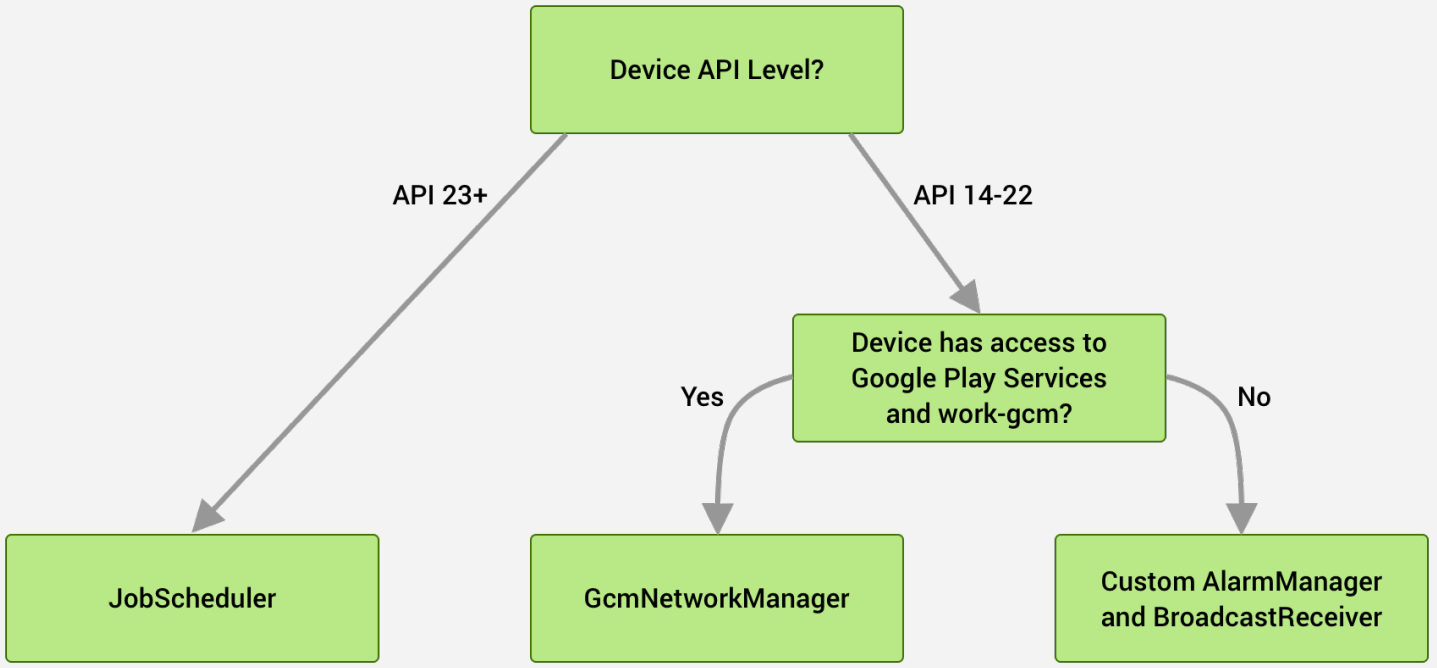


Image from: https://developer.android.com/topic/libraries/architecture/workmanager

WorkManager

- Mainly for deferrable and guaranteed tasks
  - Deferrable implies a task can be run later
  - Guaranteed implies a task should run even if the device restarts

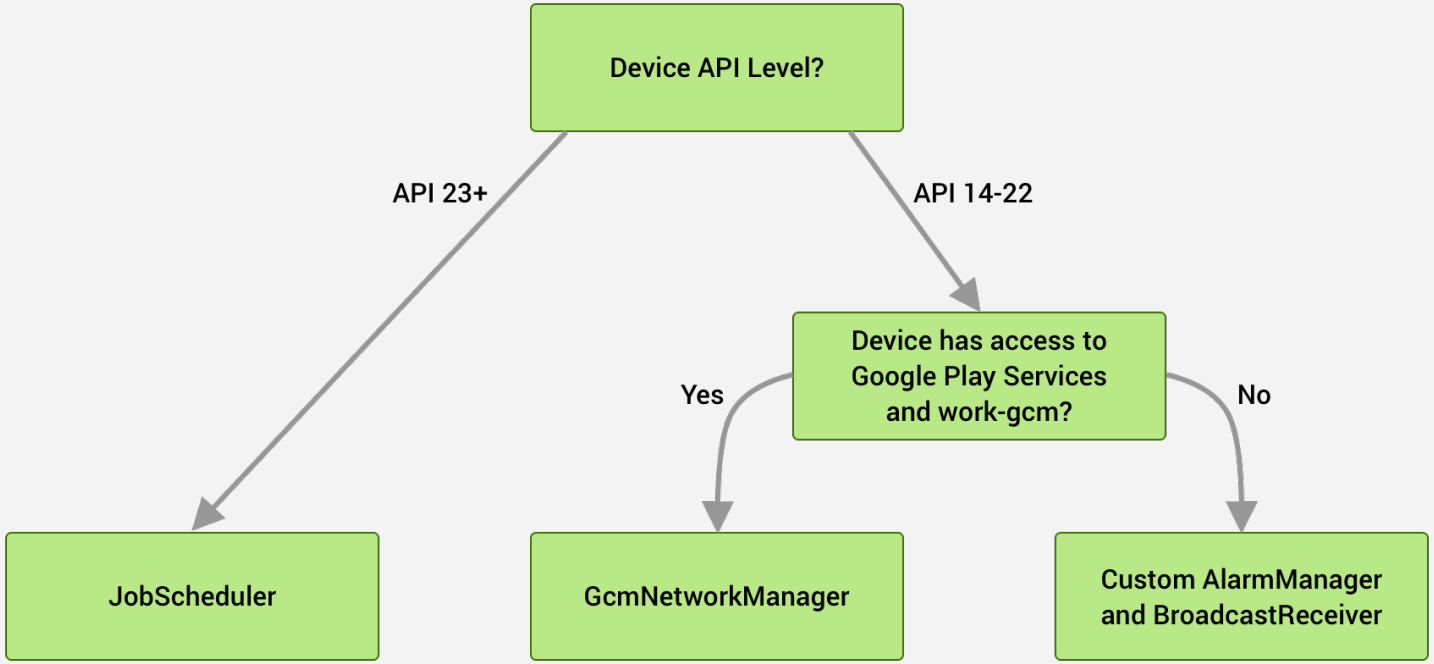


Image from: https://developer.android.com/topic/libraries/architecture/workmanager

Worker of WorkManager

```
public class MyWorker extends Worker {

    public MyWorker(Context context, WorkerParameters workerParams) {
        super(context, workerParams);
    }

    @Override
    public Result doWork() {
        // your work
        return Result.success();
    }
}
```

Requirements

Result types:

- success() – work has finished
- failure() – work failed
- retry() – work failed and should be rexecuted

WorkManager

Similar to the JobInfo of JobScheduler

```
Constraints constraints = new Constraints.Builder()
        .setRequiredNetworkType(NetworkType.UNMETERED)
        .setRequiresCharging(true)
        .build();

Data.Builder dataBuilder = new Data.Builder();
dataBuilder.putString("MyString", myString);
Data data = dataBuilder.build()

WorkRequest workRequest = new OneTimeWorkRequest.Builder(MyWorker.class)
        .setInputData(data)
        .setConstraints(constraints)
        .build();

WorkManager.getInstance(MainActivity.this)
        .enqueue(workRequest);
```

If you need to include data into the request

If you have some periodic task, you can use PeriodicWorkRequest

Your worker class where the logic is located

For more on background process limitations brought on by changes in Oreo, see the documentation:
https://developer.android.com/about/versions/oreo/background

Summary

- We mainly talked about Services as a background component
  - Not separate from the main thread, but it can be
  - We talked about the BroadcastReceiver as another means for communicating with processes
- We also talked about general background processing
  - There are a lot of APIs to choose from and there's not really a one fits all solution
  - AlarmManager, JobScheduler, WorkManager, etc.

When looking for solutions online, just be aware that a lot has changed in the past few years and that certain solutions aren't encouraged anymore

Thanks everyone!

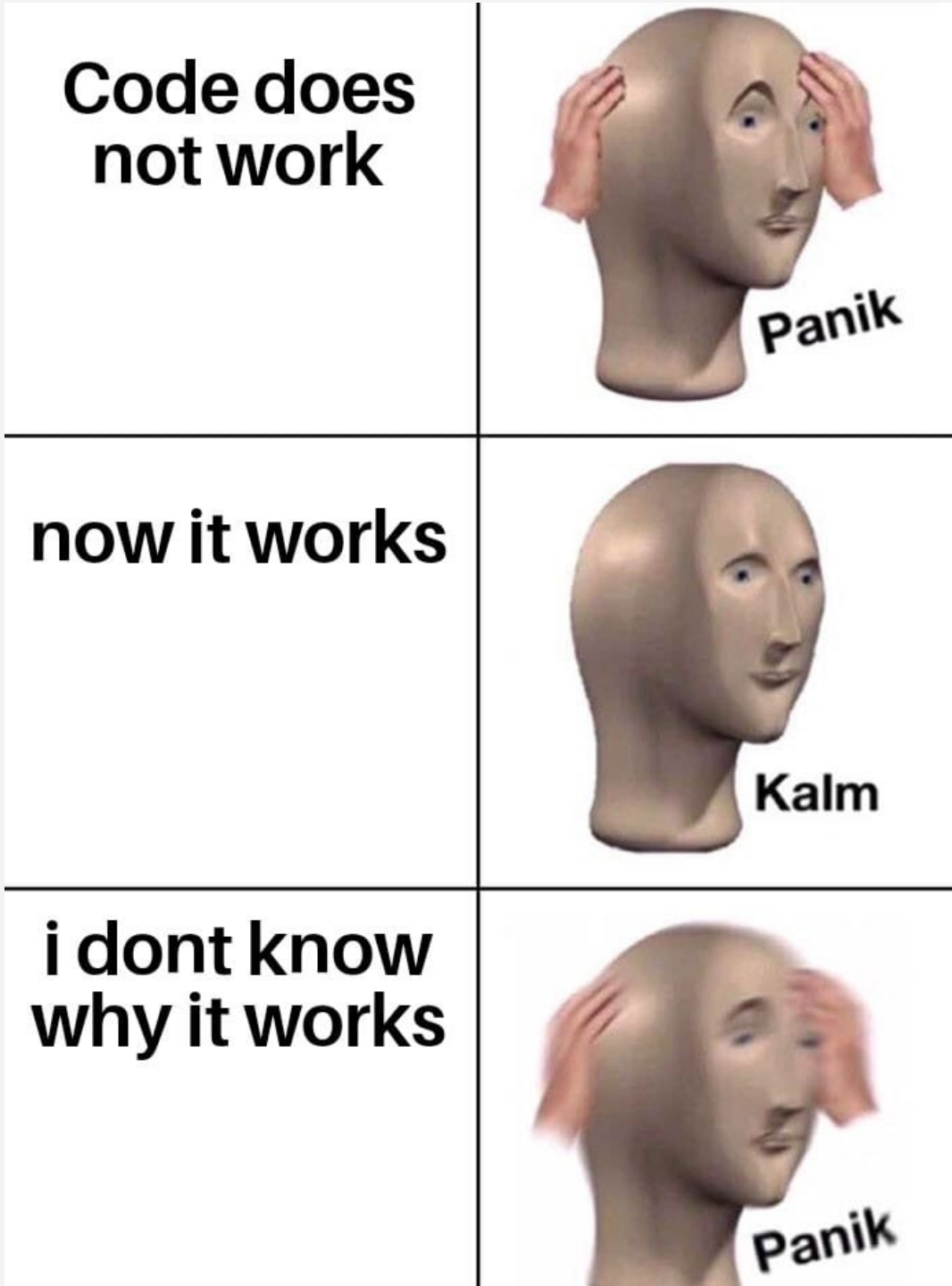


Image sauce: https://www.facebook.com/ProgrammersCreateLife/photos/a.241809332534619/3268497533199102/