Introduction to Mobile Programming

Android Programming

Chapter 6

Background Tasks

1. Every Android app has a main thread which is in charge of handling UI (including measuring and drawing views), coordinating user interactions, and receiving lifecycle events.

2. Any long-running computations and operations such as decoding a bitmap, accessing the disk, or performing network requests should be done on a separate background thread.

3. In general, anything that takes more than a few milliseconds should be delegated to a background thread.

Background Processing

- Background processing in Android refers to the execution of tasks in different threads than the Main Thread, also known as UI Thread, where views are inflated and where the user interacts with our app.

Why Background Processing?

- To avoid UI blockages by I/O events and prevent the famous *"Application Not Responding"* dialog. A freezing app means bad UX.
- Some operations are not allowed ro run in the Main Thread, such as HTTP calls.
- To improve performance.

Challenges in Background Processing

1. Background tasks consume a device's limited resources, like RAM and battery. This may result in a poor experience for the user if not handled correctly.

- Android 6.0 (API level 23) introduced Doze mode and app standby. Doze mode restricts app behavior when the screen is off and the device is stationary. App standby puts unused applications into a special state that restricts their network access, jobs, and syncs.
- Android 7.0 (API level 24) limited implicit broadcasts and introduced Doze-on-the-Go.
- Android 8.0 (API level 26) further limited background behavior, such as getting location in the background and releasing cached wakelocks.
- Android 9 (API level 28) introduced App Standby Buckets, in which app requests for resources are dynamically prioritized based on app usage patterns.

App StandBy Buckets

- Android 9 (API level 28) introduces a new battery management feature, **App Standby Buckets**.
- App Standby Buckets help the system prioritize apps' requests for resources based on how recently and how frequently the apps are used.
- Based on app usage patterns, each app is placed in one of five priority **buckets**.

Priority Buckets

- The system dynamically assigns each app to a priority bucket, reassigning the apps as needed.
- The system may rely on a preloaded app that uses machine learning to determine how likely each app is to be used, and assigns apps to the appropriate buckets.
- If the system app is not present on a device, the system defaults to sorting apps based on how recently they were used.
- More active apps are assigned to buckets that give the apps higher priority, making more system resources available to the app.
- In particular, the bucket determines how frequently the app's jobs run, how often the app can trigger alarms, and how often the app can receive high-priority Firebase Cloud Messaging messages.

1.

Priority Buckets

The buckets are:

- Active: App is currently being used or was very recently used
- Working set: App is in regular use
- Frequent: App is often used, but not every day
- Rare: App is not frequently used

In addition, there's a special **never** bucket for apps that have been installed but have never been run. The system imposes severe restrictions on these apps.

> ★ **Note:** Every manufacturer can set their own criteria for how non-active apps are assigned to buckets. You should not try to influence which bucket your app is assigned to. Instead, focus on making sure your app behaves well in whatever bucket it might be in. Your app can find out what bucket it's currently in by calling `UsageStatsManager.getAppStandbyBucket()`.

Active

Active

An app is in the **active** bucket if the user is currently using the app or very recently used the app. For example:

- The app has launched an activity
- The app is running a foreground service
- The app has a sync adapter associated with a content provider used by a foreground app
- The user clicks on a notification from the app

If an app is in the active bucket, the system does not place any restrictions on the app's jobs, alarms, or FCM messages.

Working Set

Working set

An app is in the **working set** bucket if it runs often but it is not currently active. For example, a social media app that the user launches most days is likely to be in the working set. Apps are also promoted to the working set bucket if they're used indirectly.

If an app is in the working set, the system imposes mild restrictions on its ability to run jobs and trigger alarms. For details, see Power management restrictions.

Frequent Set

Frequent

An app is in the **frequent** bucket if it is used regularly, but not necessarily every day. For example, a workout-tracking app that the user runs at the gym might be in the frequent bucket.

If an app is in the frequent bucket, the system imposes stronger restrictions on its ability to run jobs and trigger alarms, and also imposes a cap on high-priority FCM messages. For details, see Power management restrictions.

Rare

Rare

An app is in the **rare** bucket if it is not often used. For example, a hotel app that the user only runs while they're staying at that hotel might be in the rare bucket.

If an app is in the rare bucket, the system imposes strict restrictions on its ability to run jobs, trigger alarms, and receive high-priority FCM messages. The system also limits the app's ability to connect to the internet. For details, see Power management restrictions.

Best Practices

- Do not try to manipulate the system into putting your app into one bucket or another. The system's bucketing methods can change, and every device manufacturer could choose to write their own bucketing app with its own algorithm. Instead, make sure your app behaves appropriately no matter which bucket it's in.
- If an app does not have a launcher activity, it might never be promoted to the active bucket. You might want to redesign your app to have such an activity.
- If the app's notifications aren't actionable, users won't be able to trigger the app's promotion to the active bucket by interacting with the notifications. In this case, you may want to redesign some appropriate notifications so they allow a response from the user. For some guidelines, see the Material Design Notifications design patterns.
- Similarly, if the app doesn't show a notification upon receiving a high-priority FCM message, it won't give the user a chance to interact with the app and thus promote it to the active bucket. In fact, the only intended use for high-priority FCM messages is to push a notification to the user, so this situation should never occur. If you inappropriately mark an FCM message as high-priority when it doesn't trigger user interaction, it can cause other negative consequences; for example, it can result in your app exhausting its quota, causing genuinely urgent FCM messages to be treated as normal-priority.

Choosing the Right Solution for Your Work

- **Does the app need to have precise control over the start and stop time?** For example, a music app needs to start playing music when the user starts playback, and continue playing until the user stops it. By contrast, an app might need to periodically upload logs; the app wouldn't care just when the uploads happened.
- **Is the job interruptible, if necessary?** For example, if an app is doing a very large file upload, and the upload is interrupted, all the work might be lost, forcing the app to start the upload all over again. In such a case, you'd want the system to avoid interrupting the job if at all possible. By contrast, an app might periodically upload small log files, where it wouldn't be a problem if any particular upload was interrupted.
- **Can the work be deferred, or does it need to happen right away?** For example, if you need to fetch some data from the network in response to the user clicking a button, that work must be done right away. However, if you want to upload your logs to the server, that work can be deferred without affecting your app’s performance or user expectations.

Choosing the Right Solution for Your Work

- **Is the work dependent on system conditions?** You might want your job to run only when the device meets certain conditions, such as being connected to power, having internet connectivity, and so on. For example, your app might periodically need to compress its stored data. To avoid affecting the user, you would want this job to happen only when the device is charging and idle.
- **Does the work involve the collection or use of sensitive user data?** For example, if you need to provide directions in a navigation app, you can use a foreground service to continue the user-initiated action of starting navigation.
- **Does the job need to run at a precise time?** A calendar app might let a user set up a reminder for an event at a specific time. The user expects to see the reminder notification at the correct time. In other cases, the app may not care precisely when the job runs. The app might have general requirements—like, "Job A must run first, then Job B, then Job C"—but it doesn't require jobs to run at a specific time.

WorkManager

- For work that is deferrable and expected to run even if your device or application restarts, use WorkManager.
- WorkManager is an Android library that gracefully runs deferrable background work when the work's conditions (like network availability and power) are satisfied.
- WorkManager also supports running jobs as a foreground service, which is ideal when you need to do work that shouldn't be interrupted.

Foreground Services

- For user-initiated work that need to run immediately and must execute to completion, use a foreground service. Using a foreground service tells the system that the app is doing something important and it shouldn't be killed.
- Foreground services are visible to users via a non-dismissible notification in the notification tray.
- Foreground services are most appropriate when the app needs to have precise control over when the work stops and starts.
- Music App

AlarmManager

- If you need to run a job at a *precise* time, use `AlarmManager`.
  - `AlarmManager` launches your app, if necessary, to do the job at the time you specify.
- However, if your job does not need to run at a precise time, `WorkManager` is a better option;
  - `WorkManager` is better able to balance system resources.
- For example, if you need to run a job every hour or so, but *don't* need the job to run at a specific time, you should use `WorkManager` to set up a recurring job.

Services Overview

- A Service is an application component that can perform long-running operations in the background, and it doesn't 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. Additionally, a component can bind to a service to interact with it and even perform interprocess communication (IPC).
- For example, a service can handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

Types of Services

Foreground

A foreground service performs some operation that is noticeable to the user. For example, an audio app would use a foreground service to play an audio track. Foreground services must display a Notification. Foreground services continue running even when the user isn't interacting with the app.

Background

A background service performs an operation that isn't directly noticed by the user. For example, if an app used a service to compact its storage, that would usually be a background service.

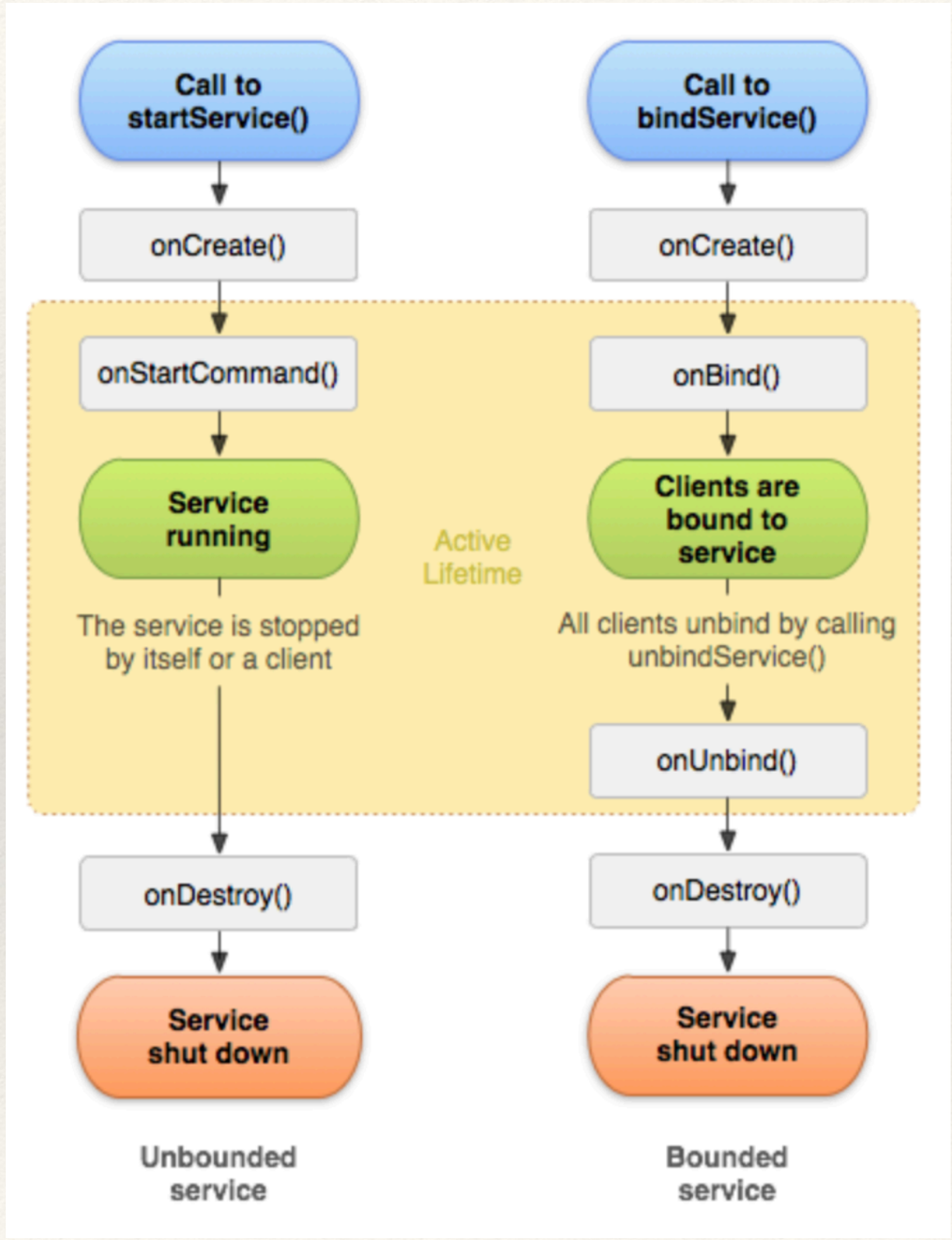
Bound

A service is *bound* when an application component binds to it by calling `bindService()`. A bound service offers a client-server interface that allows components to interact with the service, send requests, receive results, and even do so across processes with interprocess communication (IPC). A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.

Service or Thread ?

- A service is simply a component that can run in the background, even when the user is not interacting with your application, so you should create a service only if that is what you need.
- If you must perform work outside of your main thread, but only while the user is interacting with your application, you should instead create a new thread.
- For example, if you want to play some music, but only while your activity is running, you might create a thread in `onCreate()`, start running it in `onStart()`, and stop it in `onStop()`. Also consider using `AsyncTask` or `HandlerThread` instead of the traditional `Thread` class.

(Un)Bounded Services



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 intent = new Intent(this, HelloService.class);
startService(intent);
```

Stopping a Service

- aA started service must manage its own lifecycle. That is, the system doesn't stop or destroy the service unless it must recover system memory and the service continues to run after `onStartCommand()` returns.
- The service must stop itself by calling `stopSelf()`, or another component can stop it by calling `stopService()`.

Declaring a Service

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

```
public class RSSPullService extends IntentService {
    @Override
    protected void onHandleIntent(Intent workIntent) {
        // Gets data from the incoming Intent
        String dataString = workIntent.getDataString();
        ...
        // Do work here, based on the contents of dataString
        ...
    }
}
```

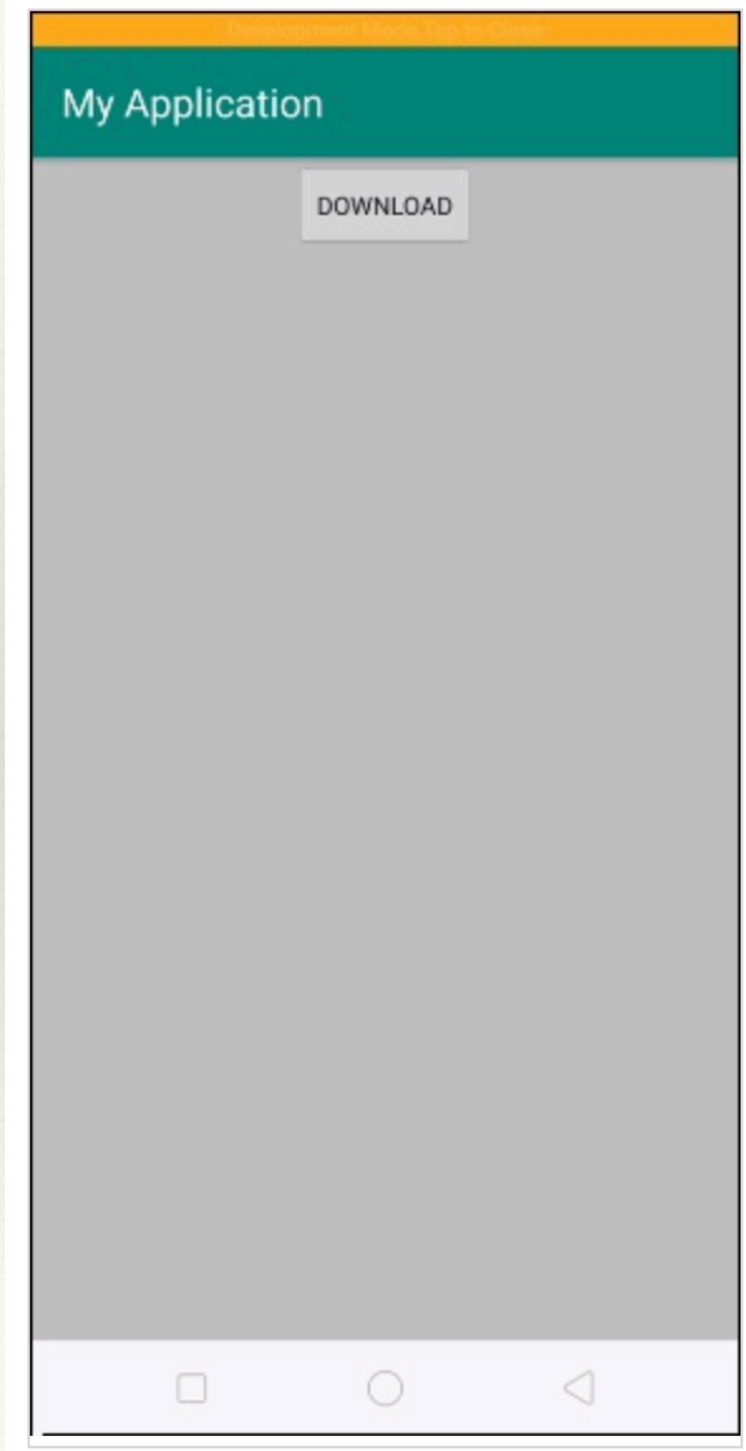
AsyncTask

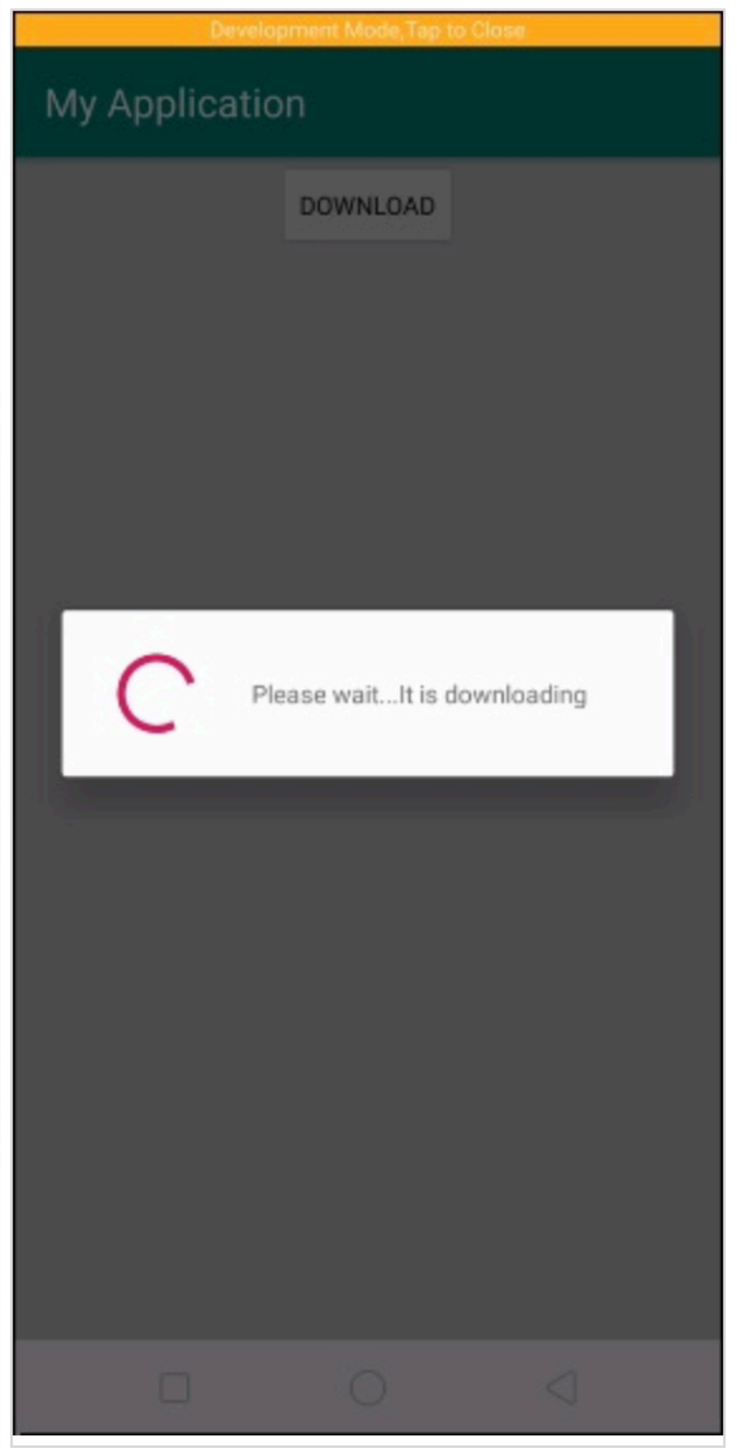
- Android AsyncTask going to do background operation on background thread and update on main thread.
- In android we cant directly touch background thread to main thread in android development.
- Asynctask help us to make communication between background thread to main thread.

Methods of AsyncTask

- **onPreExecute()** – Before doing background operation we should show something on screen like progressbar or any animation to user. we can directly comminicate background operation using on doInBackground() but for the best practice, we should call all asyncTask methods .
- **doInBackground(Params)** – In this method we have to do background operation on background thread. Operations in this method should not touch on any mainthread activities or fragments.
- **onProgressUpdate(Progress…)** – While doing background operation, if you want to update some information on UI, we can use this method.
- **onPostExecute(Result)** – In this method we can update ui of background operation result.

AsyncTask Example





```
public class MainActivity extends AppCompatActivity {
   URL ImageUrl = null;
   InputStream is = null;
   Bitmap bmImg = null;
   ImageView imageView= null;
   ProgressDialog p;
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      Button button=findViewById(R.id.asyncTask);
      imageView=findViewById(R.id.image);
      button.setOnClickListener(new View.OnClickListener() {
         @Override
         public void onClick(View v) {
            AsyncTaskExample asyncTask=new AsyncTaskExample();
            asyncTask.execute("https://www.tutorialspoint.com/images/tp-logo-di
         }
      });
   }
```

```
private class AsyncTaskExample extends AsyncTask<String, String, Bitmap> {
   @Override
   protected void onPreExecute() {
      super.onPreExecute();
      p = new ProgressDialog(MainActivity.this);
      p.setMessage("Please wait...It is downloading");
      p.setIndeterminate(false);
      p.setCancelable(false);
      p.show();
   }
   @Override
```

```
protected Bitmap doInBackground(String... strings) {
    try {
        ImageUrl = new URL(strings[0]);
        HttpURLConnection conn = (HttpURLConnection) ImageUrl.openConnection();
        conn.setDoInput(true);
        conn.connect();
        is = conn.getInputStream();
        BitmapFactory.Options options = new BitmapFactory.Options();
        options.inPreferredConfig = Bitmap.Config.RGB_565;
        bmImg = BitmapFactory.decodeStream(is, null, options);
    } catch (IOException e) {
        e.printStackTrace();
    }
    return bmImg;
}
```

```
@Override
protected void onPostExecute(Bitmap bitmap) {
    super.onPostExecute(bitmap);
    if(imageView!=null) {
        p.hide();
        imageView.setImageBitmap(bitmap);
    }else {
        p.show();
    }
}
```

Summary

AsyncTask

- AsyncTask can be good solution where your task is executed in background(worker thread) by using doInBackground() method and result is delivered to UI using onPostExecute() method.

- AsyncTask is created and executed from main thread. And you can execute instance of AsyncTask only once.

Services

- Service is best used when your task is not too long because it runs on main thread. It can be created and executed from any thread.
- If any important task need to be executed then we can use ForegroundService. ForegroundService uses notification and android system will not kill your ForegroundService.
- It blocks main thread or UI thread if started from main thread.

IntentService

- IntentService stored all requested task in queue and executes them sequentially in worker thread. It stops itself when there is no more task in its queue.

- It cannot run task in parallel and executes task after poping it from its queue

DownloadManager

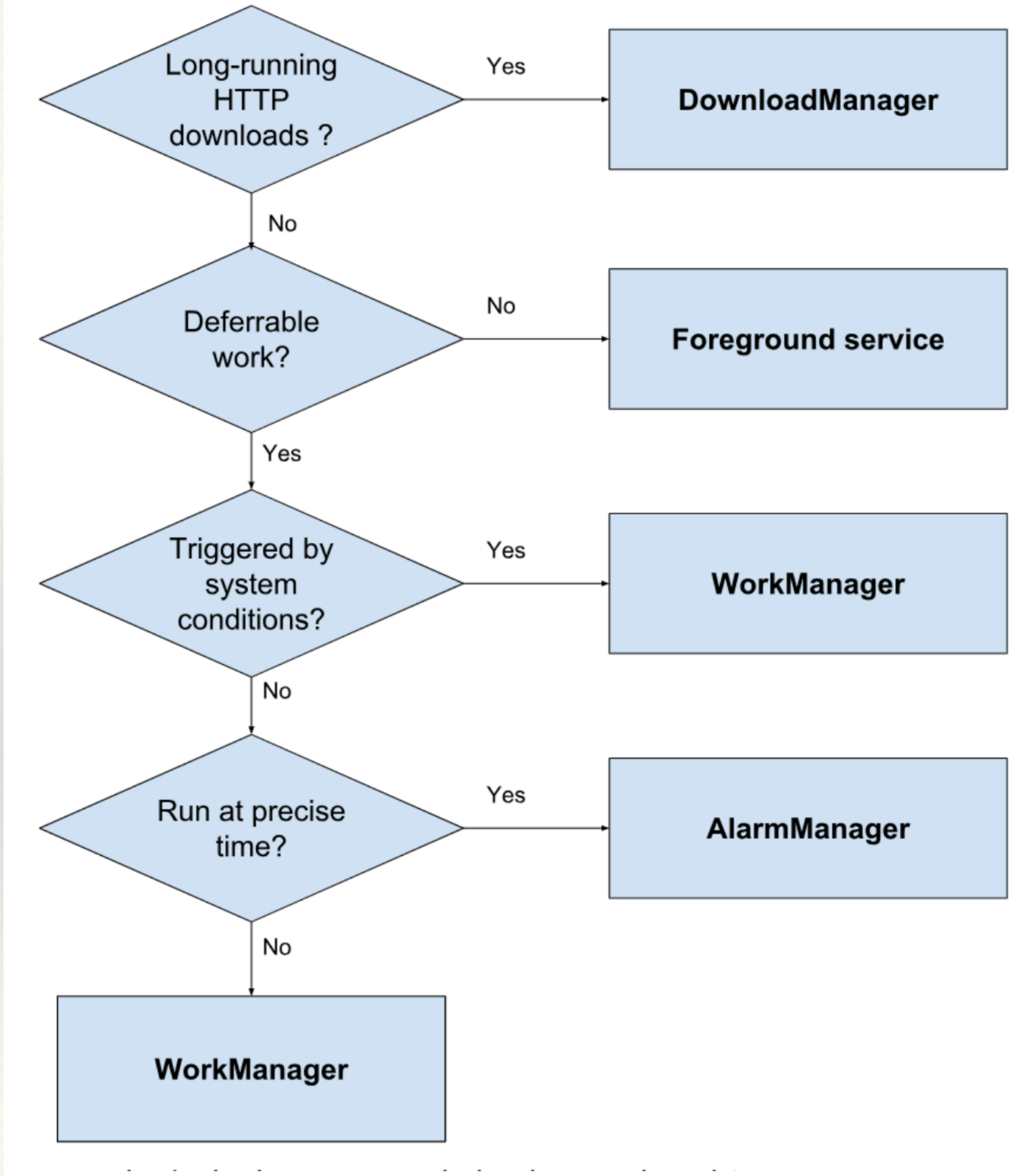
- Download manager should be preferred when app wants to download large media or document files.

WorkManager

- Latest happening in the word of android background processing is WorkManager. It is library released under Android Jetpack.
- It also takes care of system resources while executing background task. You can even execute periodic task or one time task using handy classes provided in this library.

Firebase Jobdispatcher and JobScheduler

- Jobdispatcher will execute your task in background taking into account system resources like battery and memory.
- Using Firebase jobdispatcher you can schedule your task periodically at regular interval. Internally it uses android JobScheduler to execute its task.


Long-running HTTP downloads ?
Yes
DownloadManager
No
Deferrable work?
No
Foreground service
Yes
Triggered by system conditions?
Yes
WorkManager
No
Run at precise time?
Yes
AlarmManager
No
WorkManager

```
protected Bitmap doInBackground(String... strings) {
    try {
        ImageUrl = new URL(strings[0]);
        HttpURLConnection conn = (HttpURLConnection) ImageUrl.openConnection();
        conn.setDoInput(true);
        conn.connect();
        is = conn.getInputStream();
        BitmapFactory.Options options = new BitmapFactory.Options();
        options.inPreferredConfig = Bitmap.Config.RGB_565;
        bmImg = BitmapFactory.decodeStream(is, null, options);
    } catch (IOException e) {
        e.printStackTrace();
    }
    return bmImg;
}
```

```
@Override
protected void onPostExecute(Bitmap bitmap) {
    super.onPostExecute(bitmap);
    if(imageView!=null) {
        p.hide();
        imageView.setImageBitmap(bitmap);
    }else {
        p.show();
    }
}
```