Platform-Based Development: Background Processing

BS UNI studies, Spring 2018/2019

Dr Veljko Pejović
Veljko.Pejovic@fri.uni-lj.si

University *of Ljubljana*
Faculty *of Computer and Information Science*

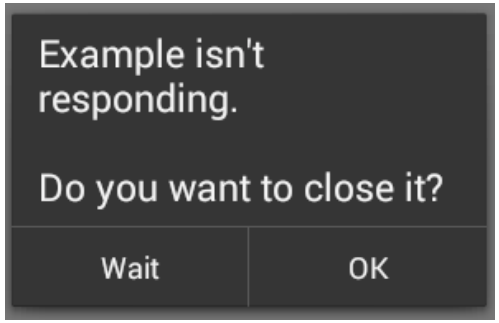
Concurrency in Android

- Java Threads
  - The most general method
- Service
  - Runs in the background, however, by default on the same thread as the UI
- IntentService
  - Background work on a separate thread, to contact the UI use local broadcast
- AsyncTask
  - Background work on a separate thread, but with a tight integration with the UI

Threads

- A UI (Main) Thread is created and started when an application is launched
  - Listens for events on UI components
  - Loops infinitely
- Your code (by default) runs on the UI Thread
- Intensive work (database access, networking) can prevent the UI thread from processing UI interaction tasks

There are no separate threads for components



Threads

- Instead, run heavy/slow operations in background threads
- Background thread processing supported by:
  - Threads + Handlers
  - IntentService
  - AsyncTask
  - Thread + Service
- Different abstractions for different goals, e.g.:
  - A music player that runs in the background
  - An online social network post button

Thread and Handlers

- Java Threads + a Handler that enables communication among the threads
- A straightforward solution:
  - Create a worker Thread
  - Put an infinite loop in it and listen for new tasks
  - De-queue the tasks, for each task:
    - Execute
    - Report results back to the UI Thread via a Handler
  - Break the loop to kill the thread

Thread and Handlers Example

Looper, Message Queue, Handler

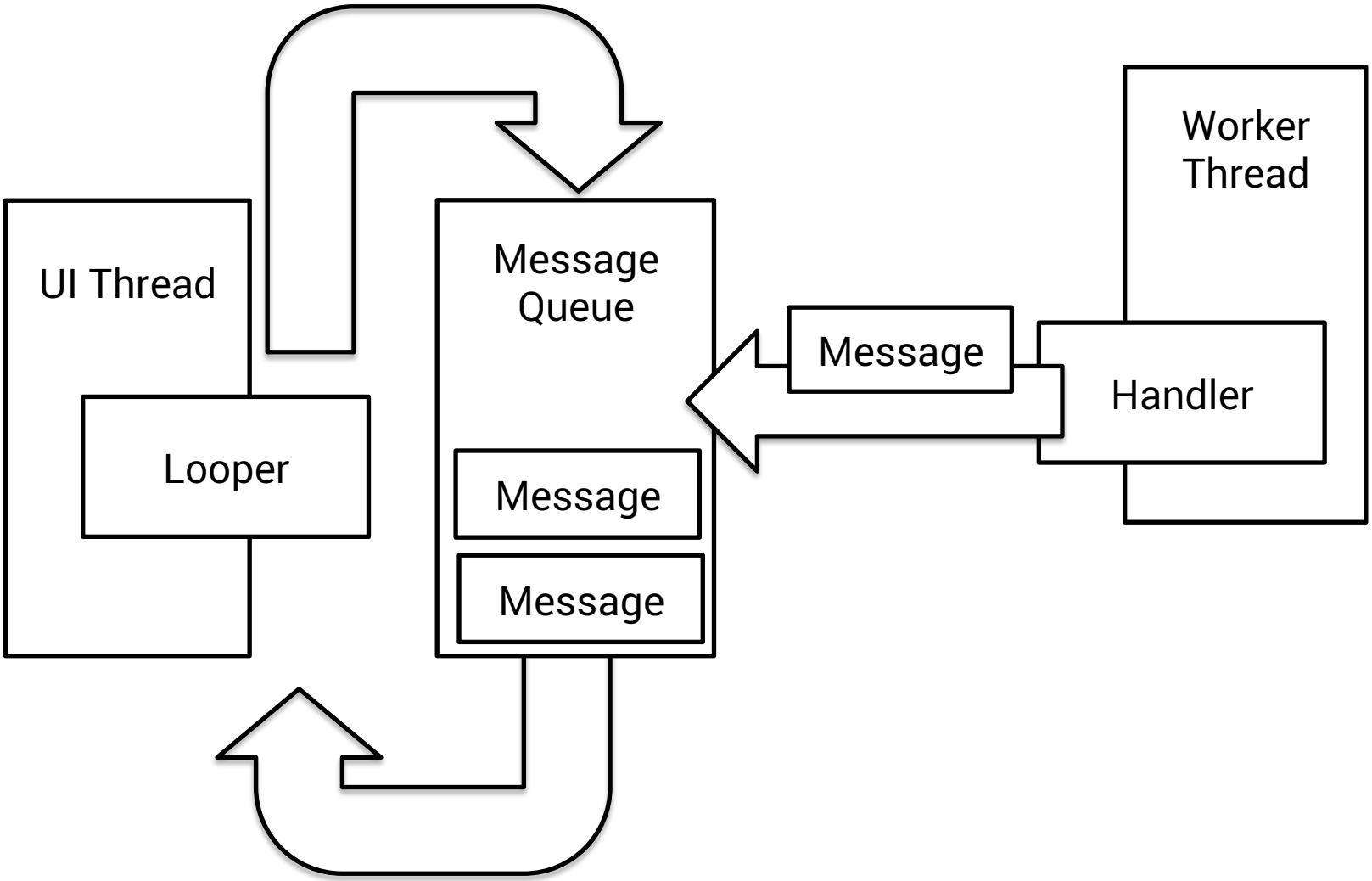
- Looper keeps the Thread alive in an infinite loop
  - Automatically created for the UI Thread
  - For custom threads, create it yourself or use HandlerThread
    - `Looper.prepare();`
    - `Looper.loop();`
    - `Looper.quit();`
- MessageQueue holds Messages/Runnables for a Thread
  - Message – for passing data to a thread
  - Runnable – a task that is executed when the thread is free (or after a predefined delay)

Looper, Message Queue, Handler

- Handler
  - Associated with a particular Thread (e.g. UI Thread)
  - Allows you to send Messages/Runnables to the MessageQueue and process them
  - Post a Message/Runnable to be handled immediately via post(Runnable r) method or after a certain delay via postDelayed(Runnable r, int msDelay)

```
new Handler(Looper.getMainLooper())
    .post(new Runnable() {
          @Override
          public void run() {
            // this will run in the main thread
          }
});
```

Thread and Handlers



HandlerThread

- A Thread that has a Looper
- Example use:
  - Instantiate a HandlerThread
  - Attach a Handler to your thread

```
HandlerThread handlerThread =
      new HandlerThread("MyHandlerThread");
handlerThread.start();
Looper looper = handlerThread.getLooper();
Handler handler = new Handler(looper);
```

Call .quit() to shut the thread down

HandlerThread Example

Services

- Activities run on the UI (main) thread and have a UI attached (layout)
  - Processing-heavy functions on the main thread impact the responsiveness
- Services can run on either the Main or separate threads and do not have a UI attached
  - Run outside UI, for long-running operations
- Services are often more convenient than custom Threads for tasks than need to be “independent” and run even when the Activity is destroyed

Background and Foreground Service

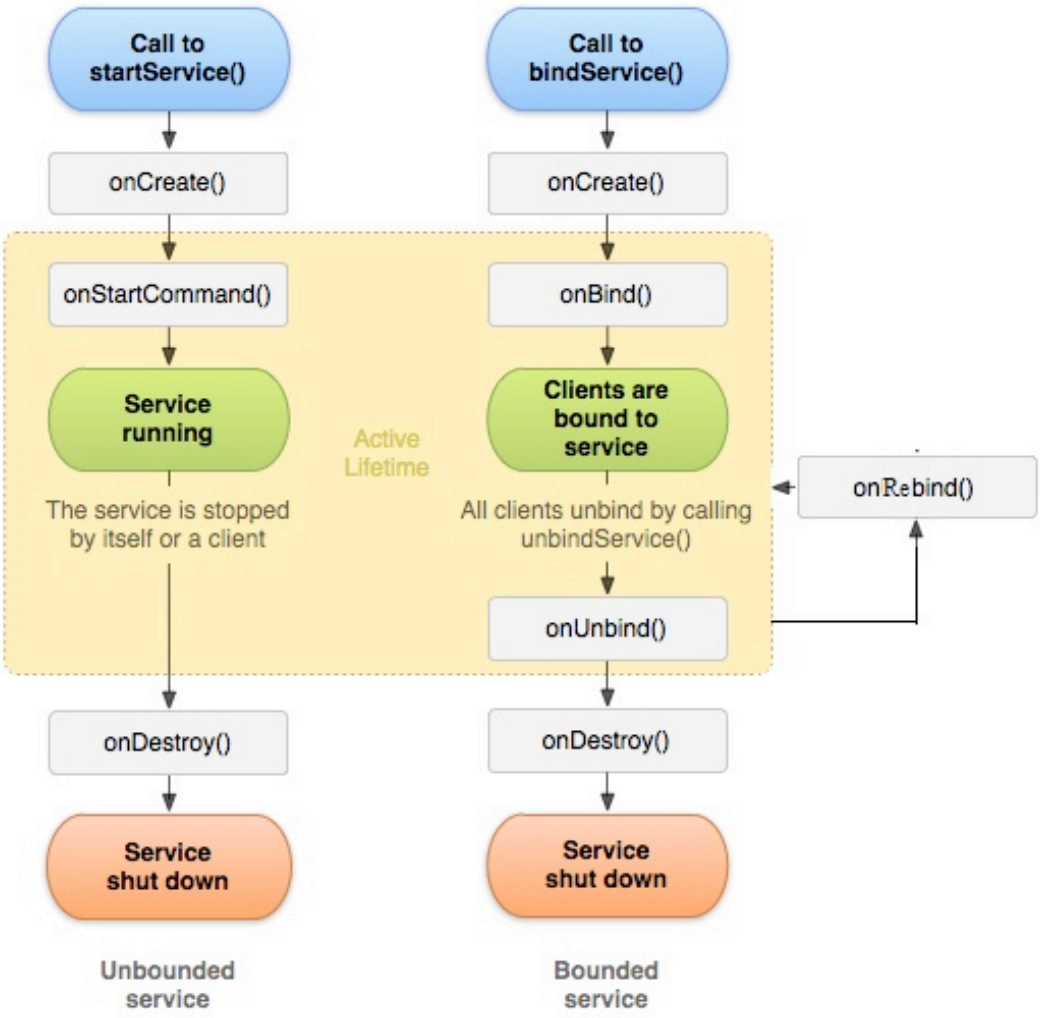
- Background Service
  - For actions that do not have to be noticed by the user (e.g. sensing a user's physical activity)
- Foreground Service
  - For actions that the user needs be aware of and should the control of (e.g. a music player app)
  - A foreground service must show a notification in the notification bar

Starting/Stopping a Service

- Services can be created:
  - Explicitly using Context.startService()
  - Implicitly, if not already running, when a client requests connection to a Service via Context.bindService()
- Services can be stopped:
  - From within the Service with stopSelf()
  - From another component with Context.stopService()

Services

- Multiple startService calls do not nest – you only have one service; however, onStartCommand() will be called repeatedly
- Service will be stopped only once with Context.stopService() or stopSelf()



Services – Bound

- Bound Services – like servers in a client-server paradigm
- Services started through binding, do not call onStartCommand()
- Return IBinder object from onBind(Intent) so that connected clients can call the Service
- The service remains running as long as the connection is established

Broadcast

- Messages sent from other components of your app, other apps or from the Android system
- Messages are wrapped in Intents

```
Intent intent = new Intent();
intent.setAction(ACTION);
intent.putExtra(STOP_SERVICE_BROADCAST_KEY, RQS_STOP_SERVICE);
sendBroadcast(intent);
```

- Send broadcasts
  - System sends certain broadcasts when an event happens, e.g. ACTION_BOOT_COMPLETED
  - Send custom broadcasts via sendBroadcast()

Broadcast

- Broadcasts are captured in an app/component if a BroadcastReceiver is registered in the code:
  - Create a BroadcastReceiver and impl. onReceive()

```
public class NotifyServiceReceiver extends BroadcastReceiver{

    @Override
    public void onReceive(Context arg0, Intent arg1) {
            …
        }
    }
}
```

  - Register for receiving certain kinds of Intents

```
IntentFilter intentFilter = new IntentFilter();
intentFilter.addAction(ACTION);
registerReceiver(notifyServiceReceiver, intentFilter);
```

Broadcast

- Broadcasts are captured in an app/component if a BroadcastReceiver is registered in AndroidManifest.XML and onReceive() is implemented in the code:

```
<receiver android:name=".MyBroadcastReceiver” android:exported="true">
 <intent-filter>
   <action android:name="android.intent.action.BOOT_COMPLETED"/>
   <action android:name="android.intent.action.INPUT_METHOD_CHANGED”/>
 </intent-filter>
</receiver>
```

```
public class MyBroadcastReceiver extends BroadcastReceiver {
     @Override
     public void onReceive(Context context, Intent intent) {
….
```

BroadcastReceiver

- Receive events announced by other components
- Events announced via Intents
  - Not the same Intent as the one starting an Activity: this one remains in the background
- Events can be announced within your app or publicly to every app on the phone
  - Announce via sendBroadcast()
- Events captured if the receiver is registered:
  - onReceiverRegistered() and then onReceive()

Service, Notification, BroadcastReceiver Example

Note on Foreground Services

- Likely to see increased use
  - Google aims to minimize background processing
  - FS for immediate guaranteed tasks, such as mobile payments, apps for unlocking garages, etc.
- In API 26 and above
  - Starting a foreground service should be done with:
  - startForegroundService() – a promise that it will go to foreground and show a notification
    followed by
  - startForeground() – the actual notification is shown

Usually in the Service's onCreate

IntentService

- A Service that
  - Runs on a separate thread
  - Queues up requests and processes them one by one
- Suitable for long running one-off tasks when we don't want to affect the UI responsiveness
- IntentService survives Activity lifecycle changes
- Called using explicit Intent
- Starts on demand, stops when it runs out of work

IntentService

- Define in AndroidManifest.XML

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

- Extend the class in your Java code

```
public class FetchAddressIntentService extends
IntentService {
```

Invoking IntentService

- Create an explicit Intent for your IntentService
- Use startService() to start the IntentService
- Add additional data if needed with the extra field

Handling Results – from IntentService to Activity (1)

- BroadcastReceiver in your Activity
  - Subclass BroadcastReceiver, implement onReceive
  - Register the receiver for a particular action for times when you would like to handle IntentService results (usually when your Activity is in the foreground)
- Broadcast from your IntentService
  - sendBroadcast() from your IS using the same Intent action as the above

Handling Results – from IntentService to Activity (2)

- ResultReceiver in your Activity
  - Subclass ResultReceiver, implement onReceiveResult

```
class AddressResultReceiver extends ResultReceiver {
    public AddressResultReceiver(Handler handler) {
        super(handler);
    }

    @Override
    protected void onReceiveResult(int resultCode,
                                   Bundle resultData) {…}
```

  - Pass ResultReceiver through Intent when starting IS

```
Intent intent = new Intent(this, FetchAddressIntentService.class);
intent.putExtra(Constants.RECEIVER, mResultReceiver);
intent.putExtra(Constants.LOCATION_DATA_EXTRA, mLastLocation);
startService(intent);
```

Handling Results – from IntentService to Activity (2)

- Set ResultReceiver result
  - IntentService sends results to ResultReceiver in a Bundle with send() method

```
Bundle bundle = new Bundle();
bundle.putString(Constants.RESULT_DATA_KEY, message);
mReceiver.send(resultCode, bundle);
```

- Example
  - Display location address

http://developer.android.com/training/location/display-address.html

IntentService Example

Based on:
https://www.vogella.com/tutorials/AndroidServices/article.html

AsyncTask

- For short, more interactive tasks
- Runs on a separate worker thread, but keeps a link with the main UI thread via:
  - onPreExecute
  - onProgressUpdate
  - onPostExecute
- Define what we want to do in the background in:
  - doInBackground
- Start with YourTask().execute()

AsyncTask <?, ?, ?>
"?" param types for input, progress, output

https://github.com/googlesamples/android-play-location/tree/master/ActivityRecognition

AsyncTask Example

```
private class PostTask extends AsyncTask<String, Integer, String> {
    @Override
    protected void onPreExecute() {
        super.onPreExecute();
        ProgressBar bar=(ProgressBar)findViewById(R.id.progressBar);
        bar.setVisibility(View.VISIBLE);
        bar.setProgress(0);
    }

 @Override
 protected String doInBackground(String... params) {
     String url=params[0];
     for (int i = 0; i <= 10; i += 1) {
         try {
             Thread.sleep(1000);
         } catch (InterruptedException e) {
             e.printStackTrace();
         }
         publishProgress(i);
     }
     return "All Done!";
 }
```

Just before the task starts

This is done in the background, and the status is communicated via publishProgress()

AsyncTask Example

```
@Override
protected void onProgressUpdate(Integer... values) {
    super.onProgressUpdate(values);
    ProgressBar bar=(ProgressBar)findViewById(R.id.progressBar);
    bar.setVisibility(View.VISIBLE);
    bar.setProgress(values[0]);
}
```

Connects with the UI thread

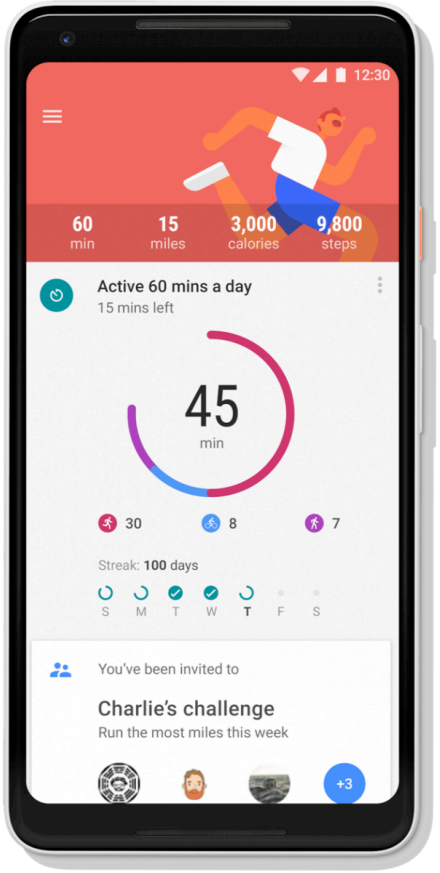
```
@Override
protected void onPostExecute(String result) {
    super.onPostExecute(result);
    ProgressBar bar=(ProgressBar)findViewById(R.id.progressBar);
    bar.setVisibility(View.GONE);
    TextView text = (TextView) findViewById(R.id.status);
    text.setText(R.string.after);
}
```

Immediately after the task is finished

AsyncTask Example

Periodic/Occasional Task Scheduling

- Numerous situations in which we require occasional processing:
  - Tracking physical activity throughout a day – e.g. Google Fit
    - Sampling sensors periodically
  - Synchronizing data with the server
    - Send data periodically, when there is WiFi connectivity
  - Reminding a user when in a particular location
    - Geofenced reminder



Periodic/Occasional Task Scheduling

- Limited battery capacity is the main issue in mobile computing
- Long and frequent background processing is the main reason for inefficient energy use:
  - Users are often unaware of background processes and their intentions, cannot easily shut them down
  - Processes consume computational and memory resources
  - Processes prevent a device from going to a low-power mode

Periodic/Occasional Task Scheduling

- Android's general direction is towards limited and controlled background processing
- In the old days (API<19):
  - schedule a periodic job to be executed every 15 mins
- Today:
  - schedule a job and Android will aggregate jobs of all apps, schedule them for a particular time slot (that you have no control off), if the app is used only rarely it might have to wait for 24 hours, and forget about getting location updates more than a few times per hour (if in background), getting notified when there is connectivity, etc.

Tools for Periodic/Occasional Task Scheduling

- Wake lock
- Foreground Service
- AlarmManager
- WorkManager (JobScheduler++)
- DownloadManager
- SyncAdapter

Wake Lock

- App prevents the phone from going to a low-power sleep mode
- Needs a special permission

```
<uses-permission android:name="android.permission.WAKE_LOCK" />
```

- Acquire a wake lock

```
PowerManager powerManager = (PowerManager)
                            getSystemService(POWER_SERVICE);
WakeLock wakeLock = powerManager
                   .newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
                                "MyApp::MyWakelockTag");
wakeLock.acquire();
```

- Release: `wakelock.release()`

This does not prevent the screen from going dark!

(use FLAG_KEEP_SCREEN_ON)

AlarmManager

- Running periodic operations at specified times or with a specified time interval
- Use when you need your tasks done at (almost) exact times between them
- Do not use for:
  - Periodic backups to the server
  - Checking for new notifications/ messages from the server

Use SyncAdapter

Use Firebase messaging if possible

AlarmManager

- Alarm types (exactness):
  - Inexact – Android will decide how to group alarms coming from multiple apps in order to optimize energy use
  - Exact – Your alarm will be executed at the prescribed time, unless the device is “sleeping”
  - Exact while idle - Your alarm will be executed at the prescribed time, even if the device is “sleeping”
- Alarm types (clock):
  - RTC – real time clock
  - ELAPSED_REALTIME – time since booted

AlarmManager

- Using AlarmManager
  - Create a BroadcastReceiver that manages the task you wish to perform when the alarm is ready
  - Set alarm
    - Define the type (exact/inexact, one off/repeating, RTC/ELAPSED)
    - Define the starting time
    - Define the repeating interval (optionally)
    - Supply Intent that starts the above BroadcastReceiver
  - Alarms can be cancelled

AlarmManager

- Restoring alarms when the device is rebooted
  - Acquire the necessary permission

```
<uses-permission
android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
```

  - Create a receiver

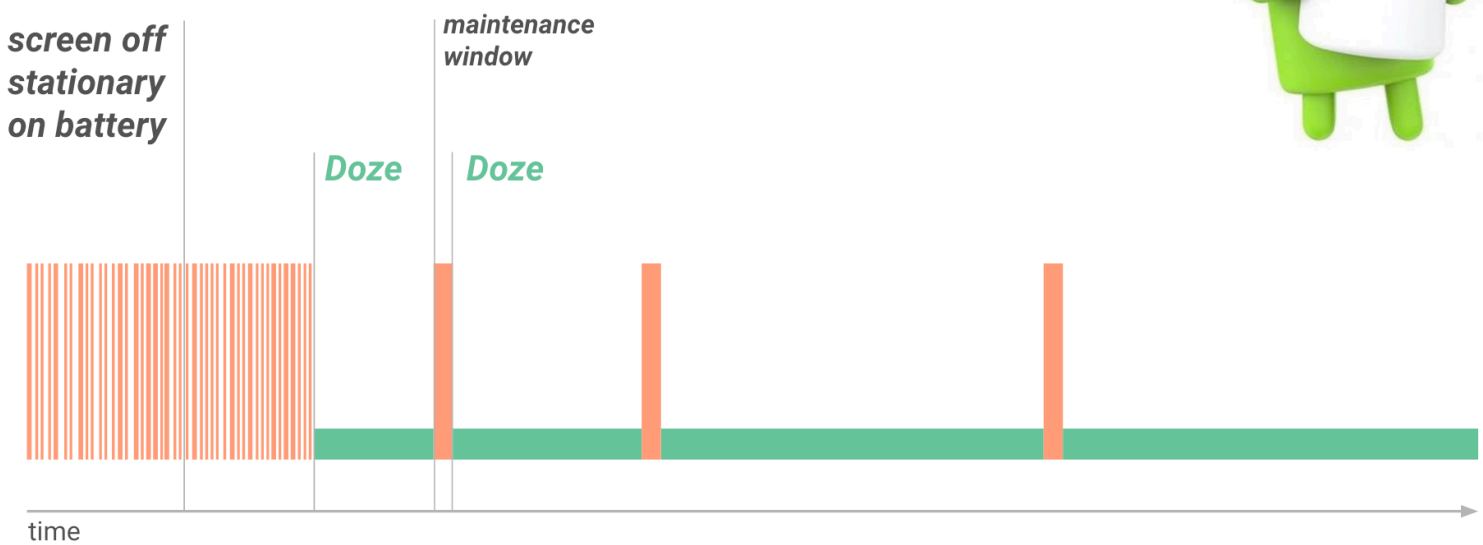
```
public class SampleBootReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        if (intent.getAction()
            .equals("android.intent.action.BOOT_COMPLETED")) {
            // Set the alarm here.
        }
```

  - Register in the manifest

```
<receiver android:name=".SampleBootReceiver”>
 <intent-filter>
   <action android:name="android.intent.action.BOOT_COMPLETED"></action>
 </intent-filter>
</receiver>
```

Doze Mode

- If a device is not charging nor actively used, it enters Doze Mode



Doze Mode

- The system sleeps most of the time
- Periodic maintenance periods when it wakes up and performs tasks from the backlog
- During the sleep time:
  - Wake locks ignored
  - Network access suspended
  - AlarmManager doesn't work
  - No WiFi scanning
  - Jobs not scheduled (see WorkManager)

unless setAndAllowWhileIdle() or setExactAndAllowWhileIdle()

Doze Mode

- To program with Doze Mode in mind, use
  - Firebase cloud messaging (FCM) for communication apps – a single connection is established
  - Use WorkManager for scheduling jobs
  - Request to be exempt from Doze
    - Can acquire partial wake lock
    - Requires a special permission

      `REQUEST_IGNORE_BATTERY_OPTIMIZATIONS`
- To test how your apps will behave when Doze Mode is active:
  - Force a device/emulator to idle mode

```
adb shell dumpsys deviceidle force-idle
```

WorkManager

- Idea:
  - Guaranteed execution
  - Constraint-aware execution
  - Respect system restrictions
  - Work without GooglePlayServices
- Implementation:
  - Part of Android Jetpack (introduced in 2018)
    - Add as a dependency to your app
  - Backwards compatible
    - Uses JobScheduler for newer APIs
    - Uses AlarmManager for older APIs

WorkManager

- Worker – a unit of work

```
public class UploadWorker extends Worker {

    public UploadWorker(
        @NonNull Context context,
        @NonNull WorkerParameters params) {
        super(context, params);
    }

    @Override
    public Result doWork() {
      // Do the work here, e.g. upload
      uploadImages()

      // Indicate whether the task finished successfully
      return Result.success()
    }
}
```

By default runs on a background thread

WorkManager

- WorkRequest – set constraints, types of execution for your work, e.g.

```
Constraints constraints = new Constraints.Builder()
    .setRequiresDeviceIdle(true)
    .setRequiresCharging(true)
     .build();

// ...then create a OneTimeWorkRequest that uses those constraints
OneTimeWorkRequest compressionWork =
                new OneTimeWorkRequest.Builder(CompressWorker.class)
     .setConstraints(constraints)
     .build();
```

WorkManager

- Running tasks

```
WorkManager.getInstance().enqueue(uploadWorkRequest);
```