Lezione 16

Programmazione Android

- Rispondere ad eventi asincroni
  - I Broadcast Receiver
- Schedulare eventi asincroni
  - Allarmi
  - WorkManager

Broadcast Receiver

Broadcast Receiver

- Dopo tutto quello che abbiamo visto su Activity e ContentProvider, i Broadcast Receiver non sorprendono
- Si tratta di una classe che ha lo scopo di ricevere (e rispondere) agli Intent inviati in broadcast
- Si estende **BroadcastReceiver**
- Si pubblica il componente in AndroidManifest.xml
  - Con i relativi intent filter
  - Il BroadcastReceiver può anche essere privato
    - Ma in questo caso, meglio usare LocalBroadcastReceiver

Attributi di <receiver>

Attribute	Description
enabled	Specify whether the receiver is enabled or not (that is, can be instantiated by the system).
exported	Flag indicating whether the given application component is available to other applications.
icon	A Drawable resource providing a graphical representation of its associated item.
label	A user-legible name for the given item.
name	Required name of the class implementing the receiver, deriving from BroadcastReceiver.
permission	Specify a permission that a client is required to have in order to use the associated object.
process	Specify a specific process that the associated code is to run in.

Più alcuni altri di uso più raro: logo, banner, description, singleUser

Ciclo di vita di un BroadcastReceiver

- I BR hanno un ciclo di vita semplicissimo
  - Il BroadcastReceiver viene creato (istanziato) quando c'è bisogno di lui
  - L'Intent viene passato al suo metodo **onReceive**()
  - Al ritorno da onReceive(), l'oggetto viene rilasciato
- Conseguenze
  - **onReceive**() non può fare molto: non dura
  - Può però invocare un Service (vedremo come), o lanciare task asincroni (senza risultato)
  - Generalmente, meglio **non** lanciare un'activity
    - Ma ci sono eccezioni, es: telefonata in arrivo → parte il dialer

Ciclo di vita di un BroadcastReceiver

- La onReceive() viene eseguita normalmente **nel thread della UI**
  - Quindi, non lo si può bloccare a lungo
  - Per sicurezza, il sistema uccide l'applicazione se la onReceive() dura più di 10 secondi
  - In realtà, qualunque cosa oltre i 0.5s è una TRAGGEDIA
- Se è stato chiamato sendOrderedBroadcast() per inviare l'Intent, è possibile impostare un risultato via setResultCode()
  - Dettagli fra poco

Pattern per i broadcast asincroni

- Molti servizi di sistema che inviano delle notifiche **broadcast** finiscono per essere gestiti da BroadcastReceiver
  - Anche un'activity può essere attivata da un Intent
  - Ma è molto brutto per eventi asincroni (si interrompe l'utente)
    - Brutto: arriva un SMS, e parte l'Activity di messaggistica
    - Meglio: arriva un SMS, parte un Broadcast Receiver, il quale posta una notifica nella Notification Area, che poi quando selezionata dall'utente fa partire l'Activity di messaggistica

Esempio

- La dichiarazione di un BR in AndroidManifest.xml include (praticamente) sempre degli intent filter
  - Non è a rigore indispensabile
  - Il BR potrebbe rispondere solo a intent espliciti
    - Ma in questo caso, meglio LocalBroadcastReceiver, no?!

```
<receiver android:name="CallReceiver">
  <intent-filter>
    <action android:name="android.intent.action.PHONE_STATE">
    </action>
  </intent-filter>
</receiver>
```

Esempio

- Implementazione di CallReceiver:

```
public class CallReceiver extends BroadcastReceiver {

  @Override
  public void onReceive(Context context, Intent intent) {
    Bundle extras = intent.getExtras();
    if (extras != null) {
      String state = extras.getString(TelephonyManager.EXTRA_STATE);
      if (state.equals(TelephonyManager.EXTRA_STATE_RINGING)) {
        String phoneNumber =
           extras.getString(TelephonyManager.EXTRA_INCOMING_NUMBER);
        Log.d("CALL", phoneNumber);
      }
    }
  }
}
```

Quali Intent?

- Non c'è una lista completa degli Intent che possono essere ricevuti
  - Ogni app può definire i propri
- Gli Intent che *il sistema* invia in broadcast sono documentati
  - … male...

<SDK>/platforms/android-21/data/broadcast_actions.txt

```
android.app.action.ACTION_PASSWORD_CHANGED
android.app.action.ACTION_PASSWORD_EXPIRING
android.app.action.ACTION_PASSWORD_FAILED
android.app.action.ACTION_PASSWORD_SUCCEEDED
android.app.action.DEVICE_ADMIN_DISABLED
android.app.action.DEVICE_ADMIN_DISABLE_REQUESTED
android.app.action.DEVICE_ADMIN_ENABLED
android.app.action.LOCK_TASK_ENTERING
android.app.action.LOCK_TASK_EXITING
android.app.action.NEXT_ALARM_CLOCK_CHANGED
android.app.action.PROFILE_PROVISIONING_COMPLETE
android.bluetooth.a2dp.profile.action.CONNECTION_STATE_CHANGED
android.bluetooth.a2dp.profile.action.PLAYING_STATE_CHANGED
android.bluetooth.adapter.action.CONNECTION_STATE_CHANGED
android.bluetooth.adapter.action.DISCOVERY_FINISHED
android.bluetooth.adapter.action.DISCOVERY_STARTED
android.bluetooth.adapter.action.LOCAL_NAME_CHANGED
android.bluetooth.adapter.action.SCAN_MODE_CHANGED
android.bluetooth.adapter.action.STATE_CHANGED
android.bluetooth.device.action.ACL_CONNECTED
android.bluetooth.device.action.ACL_DISCONNECTED
android.bluetooth.device.action.ACL_DISCONNECT_REQUESTED
android.bluetooth.device.action.BOND_STATE_CHANGED
android.bluetooth.device.action.CLASS_CHANGED
android.bluetooth.device.action.FOUND
android.bluetooth.device.action.NAME_CHANGED
android.bluetooth.device.action.PAIRING_REQUEST
android.bluetooth.device.action.UUID
android.bluetooth.devicepicker.action.DEVICE_SELECTED
android.bluetooth.devicepicker.action.LAUNCH
android.bluetooth.headset.action.VENDOR_SPECIFIC_HEADSET_EVENT
android.bluetooth.headset.profile.action.AUDIO_STATE_CHANGED
android.bluetooth.headset.profile.action.CONNECTION_STATE_CHANGED
android.bluetooth.input.profile.action.CONNECTION_STATE_CHANGED
android.bluetooth.pan.profile.action.CONNECTION_STATE_CHANGED
android.hardware.action.NEW_PICTURE
android.hardware.action.NEW_VIDEO
android.hardware.hdmi.action.OSD_MESSAGE
...
```

Quali Intent?

- Ogni versione di Android si sente libera di variare gli Intent di sistema inviati in broadcast
- Per esempio, da Android 7.0+:
  - ACTION_NEW_PICTURE e ACTION_NEW_VIDEO non vengono più inviati
  - CONNECTIVITY_ACTION viene inviato solo ai BR registrati *dinamicamente*, e non a quelli che lo indicano in un tag <filter> nel Manifest
- Da Android 8.0+:
  - La **maggior parte** dei broadcast di sistema *impliciti* viene inviata solo a BR registrati *dinamicamente*
- Cosa non si fa per la batteria!

https://developer.android.com/guide/components/broadcast-exceptions

Registrazione dinamica di un BroadcastReceiver

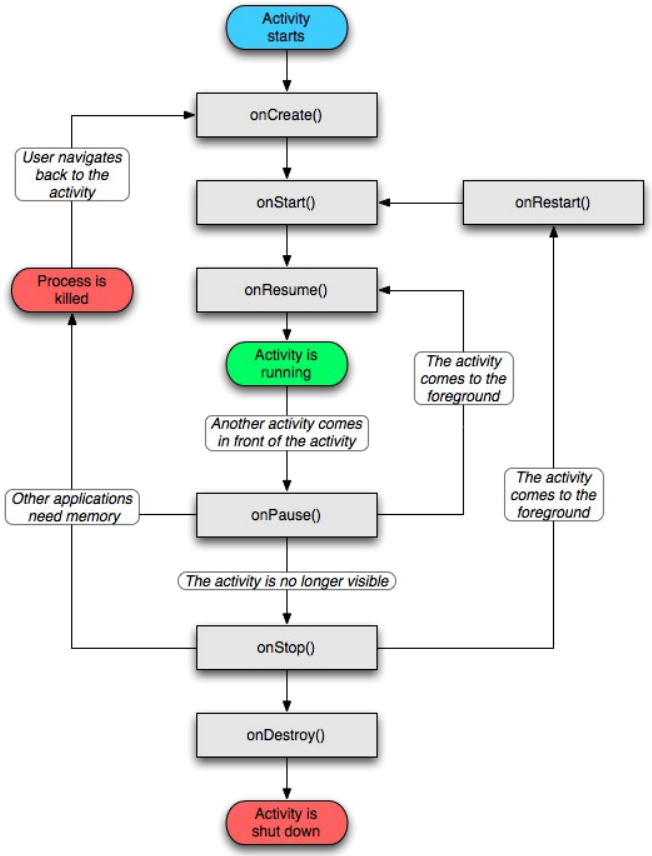
- È anche possibile registrare e deregistrare dinamicamente un BroadcastReceiver
  - In questo modo, l'applicazione riceve broadcast solo quando il receiver è registrato
- **public abstract** Intent registerReceiver (BroadcastReceiver receiver, IntentFilter filter)
- **public abstract** void unregisterReceiver (BroadcastReceiver receiver)
- Entrambi sono metodi di Context

Registrazione dinamica di un BroadcastReceiver

- Alcuni Intent inviati in broadcast possono essere definiti *sticky*
  - Dopo essere stati inviati in broadcast a tutti i receiver del sistema il cui IntentFilter corrisponde all'Intent sticky, rimangono “vivi”
  - Se successivamente al broadcast si registra dinamicamente un nuovo BroadcastReceiver che corrisponde,
    - L'Intent sticky viene inviato normalmente al nuovo receiver
    - **E** viene anche restituito dalla registerReceiver()
    - Se il parametro receiver è **null**, viene solo restituito

Registrazione dinamica di un BroadcastReceiver

- La registrazione dinamica si usa spesso quando il BroadcastReceiver è utile solo nel contesto di un'Activity
  - In questo caso, la registrazione deve rispettare il ciclo di vita dell'Activity, es.:
    - Registrazione in onResume()
    - Deregistrazione in onPause()
- Sono possibili altri schemi
  - … con cautela!



Invio di Intent in broadcast

- Nella maggior parte dei casi, le applicazioni saranno interessate a ricevere i broadcast di sistema
  - Quelli di sistema sono ben noti
  - Broadcast “privati” di un'app richiedono di conoscere l'app e la struttura/semantica dell'Intent
- Comunque, si possono pur sempre inviare propri Intent in broadcast
  - public abstract void sendBroadcast (Intent intent)
    - metodo di Context, come al solito

Invio di Intent in broadcast

- **Importante!**
  - sendBroadcast(i) → invia i a **tutti** i **BroadcastReceiver** corrispondenti
  - startActivity(i) → invia i a **una Activity** corrispondente
  - L'intent è lo stesso, i meccanismi di dispatch sono diversi!
- La chiamata a sendBroadcast() è *asincrona*
  - Ritorna immediatamente al chiamante
  - Nel frattempo il sistema, con calma, manda gli Intent

Varianti per il broadcast Permessi

- Invia solo a chi ha dichiarato un certo permesso
  - public abstract void sendBroadcast (Intent intent, String receiverPermission)
  - Solo i BroadcastReceiver che fanno parte di una app che ha richiesto (e ricevuto) il permesso riceveranno l'intent
  - Il permesso può essere una stringa custom
  - Come al solito, va richiesta con <uses-permission>

Varianti per il broadcast Serializzazione

- Invio serializzato e ordinato
  - Normalmente, i BroadcastReceiver di app diverse vengono eseguiti concorrentemente
  - È possibile chiedere invece l'invio serializzato e ordinato (in base al valore di android:priority del receiver)
  - public abstract void sendOrderedBroadcast (Intent intent, String receiverPermission)

Varianti per il broadcast Abort

- In risposta a un sendOrderedBroadcast(), i receiver possono anche **abortire** il broadcast e restituire **risultati**
- Per controllare l'abort anzitempo:
  - public final void abortBroadcast()
  - public final void clearAbortBroadcast()
  - public final boolean getAbortBroadcast()
- Il valore del flag all'uscita dalla onReceive() determina se il broadcast deve continuare

Varianti per il broadcast Risultati

- In risposta a un sendOrderedBroadcast(), i receiver possono anche **abortire** il broadcast e restituire **risultati**
- Per avere dei risultati, si invia l'intent con
  - public abstract void sendOrderedBroadcast (
    Intent intent,
    String receiverPermission,
    BroadcastReceiver resultReceiver,
    Handler scheduler,
    int initialCode,
    String initialData,
    Bundle initialExtras)
  - Si attiva un *folding* con resultReceiver in fondo

Varianti per il broadcast Risultati

- Il processo di *folding* consiste nel passare a ogni receiver il risultato (cumulato) dei receiver chiamati prima di lui
- Ciascun receiver può leggere il risultato dei predecessori, e impostare il proprio
- Il primo receiver della catena riceve i valori indicati dalla sendOrderedBroadcast()
- L'ultimo receiver della catena riceve il risultato (cumulato) lasciato dal penultimo

Varianti per il broadcast Risultati

- Il corpo di onReceive() può leggere i risultati parziali del *fold*
  - getResultCode(), getResultData(), getResultExtra()
- … e impostare i risultati parziali per il prossimo receiver nella catena
  - SetResultCode(), setResultData(), setResultExtra()
  - setResult(int code, String data, Bundle extras)
- Validi solo se siamo in un ordered broadcast
  - isOrderedBroadcast() restituisce true

Varianti per il broadcast Sticky

- Come abbiamo detto, è possibile lanciare Intent classificati come *sticky* (permanenti)
  - public abstract void sendStickyBroadcast(Intent intent)
  - public abstract void sendStickyOrderedBroadcast(Intent intent, BroadcastReceiver resultReceiver, Handler scheduler, int initialCode, String initialData, Bundle initialExtras)
- Il receiver può usare un metodo per sapere se l'intent che sta processando è uno *sticky* rimasto indietro, oppure se è un intent “fresco”
  - public final boolean isInitialStickyBroadcast()

Varianti per il broadcast Sticky

- Gli Intent inviati come *sticky* sono mantenuti dal sistema in una cache
  - Così sono subito disponibili a componenti abilitati in seguito
    - BroadcastReceiver statici, al momento in cui parte l'app
    - BroadcastReceiver dinamici, al momento della registrazione
- Il mittente può esplicitamente togliere un Intent lanciato in precedenza dalla cache
  - public void removeStickyBroadcast(Intent intent)

Varianti per il broadcast Sticky – esempio

- Per usare intent *sticky*, le app devono avere il permesso BROADCAST_STICKY
- In effetti, sono tipicamente usati (solo) dal sistema
- Per esempio:
  - Il BatteryManager invia un intent *sticky* per indicare il livello di carica corrente della batteria (e anche lo stato di ricarica o meno)
  - In questo modo, qualunque applicazione può recuperare il più recente Intent di questo tipo inviato

Varianti per il broadcast Sticky – esempio

- Leggere lo stato di carica corrente

```
IntentFilter ifilter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
Intent battery = context.registerReceiver(null, ifilter);
```

Non registriamo un receiver: prendiamo l'ultimo intent sticky come risultato

```
int status = battery.getIntExtra(BatteryManager.EXTRA_STATUS, -1);
status == BatteryManager.BATTERY_STATUS_CHARGING → in carica
status == BatteryManager.BATTERY_STATUS_FULL; → carica completa

int chargePlug = battery.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);
chargePlug == BATTERY_PLUGGED_USB; → in carica via cavo USB
chargePlug == BATTERY_PLUGGED_AC; → in carica via alimentatore
```

Broadcast locali

- In rari casi, può essere utile inviare Intent in broadcast solo all'interno della propria app
- In questo caso, la classe LocalBroadcastManager fornisce alcuni metodi di utilità
  - Si evita il costo dell'IPC e della serializzazione

static LocalBroadcastManager getInstance(Context context)

void registerReceiver(BroadcastReceiver receiver, IntentFilter filter)

boolean sendBroadcast(Intent intent)

void sendBroadcastSync(Intent intent)

void unregisterReceiver(BroadcastReceiver receiver)

Alarm

AlarmManager

- Fra i molti servizi di sistema di Android, uno si occupa di impostare ed inviare **Allarmi**
- L'invio di un allarme è realizzato tramite l'invio di un Intent (esplicito) al componente che ha registrato l'alarm
- Si ottiene il puntatore all'AlarmManager invocando getSystemService()

```
AlarmManager am =
    (AlarmManager)getSystemService(Context.ALARM_SERVICE)
```

Impostare un allarme

am.set(int type, long triggerAtTime, PendingIntent operation)

- Un allarme è definito da un *tipo* e da un *tempo*
- Viene fornito anche il PendingIntent da lanciare quando scatterà l'allarme
- Il tipo può essere:
  - ELAPSED_REALTIME – tempo dal boot
  - ELAPSED_REALTIME_WAKEUP – tempo dal boot, ma in più sveglia il dispositivo se è in sleep
  - RTC – real time clock
  - RTC_WAKEUP – real time clock, ma sveglia il dispositivo se è in sleep

Impostare un allarme

am.setRepeating(int type, long triggerAtTime, long interval, PendingIntent operation)

- Imposta un allarme con ripetizione
- Verrà inviato l'intent al *triggerAtTime*, e poi ogni *interval* (finché non viene cancellato)
- Intervalli predefiniti:
  - INTERVAL_DAY, INTERVAL_HALF_DAY, INTERVAL_HOUR, INTERVAL_FIFTEEN_MINUTES

Impostare un allarme

am.setInexactRepeating(int type, long triggerAtTime, long interval, PendingIntent operation)

- Imposta un allarme con ripetizione approssimata
- L'intervallo è (più o meno) garantito
- L'allineamento no
  - Tipicamente, il sistema cerca di “allineare” le sveglie in modo da fare il wakeup una sola volta

Cancellare un allarme

am.cancel(PendingIntent operation)

- Cancella tutti gli allarmi registrati con l'Intent passato

Alternative ad Alarm

- Usare gli Alarm ha senso quando si vuole essere svegliati a un dato momento
  - Anche se la vostra applicazione non è in esecuzione al momento!
- Per compiti di temporizzazione pura, meglio usare altri costrutti
  - es.: quelli nativi di Java: Thread.sleep(), System.currentTimeMillis()
  - oppure, usare un Handler con postDelayed()

Alarm e Wake Lock

- Gli alarm scattano (ovviamente) anche quando il telefono è in *sospensione* (sleep)
- Durante la onReceive() di chi ha ricevuto il PendingIntent, il telefono viene risvegliato e tenuto sveglio
- … ma al ritorno dalla onReceive(), può tornare immediatamente in sleep
  - Se la onReceive() ha fatto partire qualche attività asincrona, questa potrebbe fermarsi immediatamente perché il telefono torna in sospensione

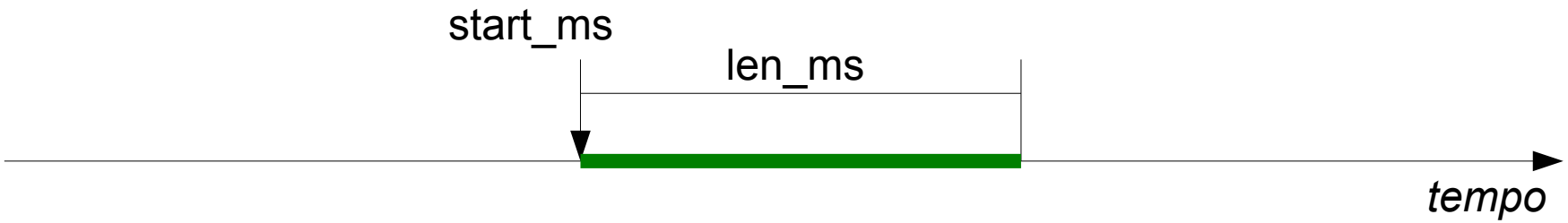
Modifiche ad Alarm in Android 4.0+

- Nelle versioni di Android precedenti KitKat, gli allarmi erano sempre esatti
  - A meno che non fossero richiesti esplicitamente inesatti
- Da Android 4 in poi, se il vostro target è API 19 o successivo, gli allarmi sono per default inesatti
  - Il sistema si sente libero di “aggiustarli” per minimizzare il numero di risvegli (e il consumo di batteria)

Modifiche ad Alarm in Android 4.0+

- Le applicazioni possono controllare la quantità di “aggiustamento” concessa al sistema

  am.setWindow (int type, long start_ms, long len_ms,
  PendingIntent operation)



- L'allarme potrà scattare in qualunque punto della finestra concessa

Modifiche ad Alarm in Android 4.0+

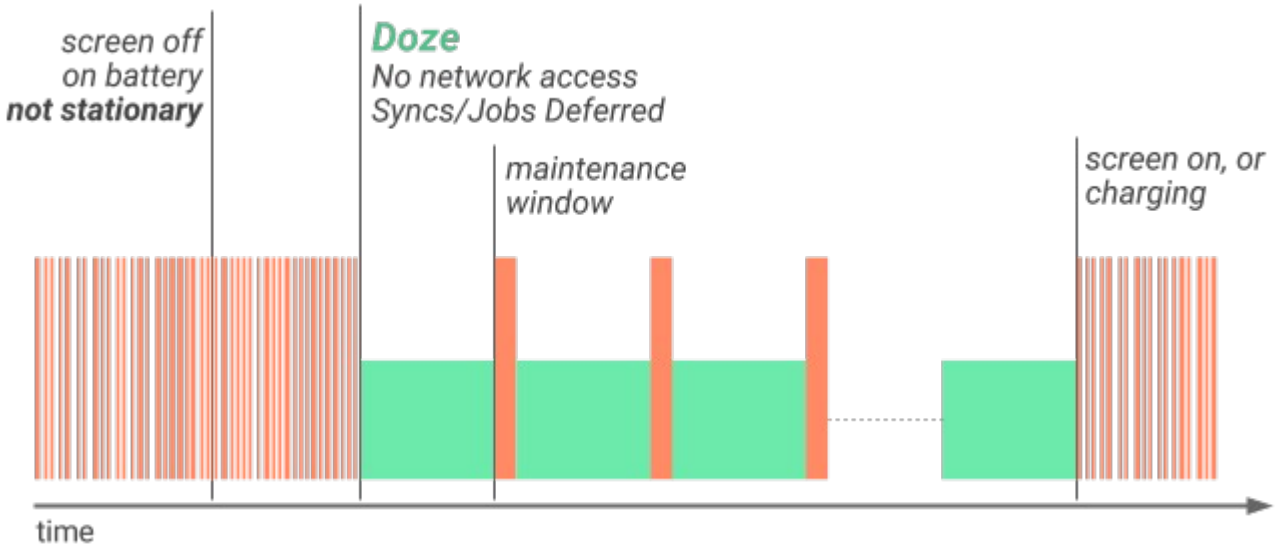
- Visto che am.set() può ora aggiustare l'allarme, serve un modo per chiederlo esatto

- In pratica, è il comportamento della set() nelle versioni di Android vecchie

```
am.setExact(int type, long trigger, PendingIntent operation)
```

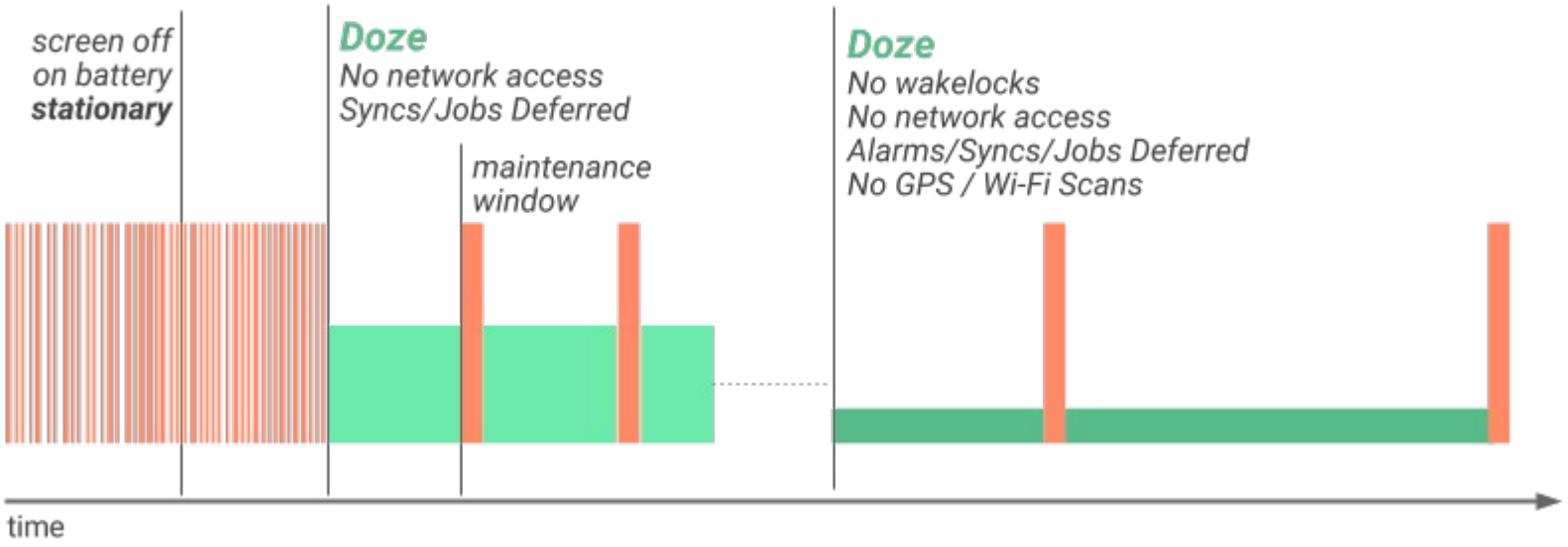
Modifiche ad Alarm in Android 7.0+

- La nuova modalità *Doze* può ritardare l'attivazione degli allarmi
- Con schermo spento e **device in movimento** gli allarmi funzionano normalmente:



Modifiche ad Alarm in Android 7.0+

- La nuova modalità *Doze* può ritardare l'attivazione degli allarmi
- Con schermo spento e **device stazionario per un certo tempo** gli allarmi possono essere ritardati:



Allarmi dalla lunga vita

- Gli allarmi pendenti vengono annullati con lo shutdown del dispositivo
  - Molto poco adatto, per esempio, per una sveglia!
- La ri-abilitazione va fatta a mano:
  - Si registra *staticamente* un BroadcastReceiver per l'intent ACTION_BOOT_COMPLETED
    - Richiede il permesso RECEIVE_BOOT_COMPLETED, ottenuto staticamente
  - Nella onReceive(), si recuperano i dati delle sveglie (per esempio, da un DB) e si re-impostano gli allarmi

Note su Doze

- Quando il sistema è in modo Doze:
  - Non si ha accesso alla rete
  - Vengono ignorati i Wake Lock
  - Gli allarmi dell'AlarmManager possono essere posticipati
    - Alla successiva maintainance window
  - Non viene fatto lo scan delle reti wi-fi
  - Non vengono eseguiti i sync adapter e i task del JobScheduler
    - Sul JobScheduler diremo di più la prossima lezione

Note su Doze

- È possibile chiedere esplicitamente di impostare degli allarmi **critici** che superano il Doze:
  - am.setAndAllowWhileIdle(int type, long trigger, PendingIntent operation)
  - am.setExantAndAllowWhileIdle(int type, long trigger, PendingIntent operation)
- Usare questi allarmi, ovviamente, peggiora la durata della batteria
  - … e state sicuri che gli utenti **vi scopriranno!**

WorkManager

WorkManager

- WorkManager è una delle (tante) API di Jetpack, dedicata all'esecuzione affidabile di **task asincroni deferrable coordinati** e **vincolati**
  - **Asincrono** = viene eseguito in qualche punto del futuro
  - **Deferrable** = può essere rimandato senza danno
  - **Coordinati** = si possono esprimere strutture (task in sequenza, in parallelo, ecc.), vd. farm, pipeline, map/reduce, e passare mappe chiavi/valori fra task
  - **Vincolati** = per ogni task, si possono specificare condizioni che devono essere verificate per l'esecuzione

Work

- Un task da eseguire è definito da un'istanza di **Worker**
  - Come al solito, definiremo una sottoclasse
  - Un solo metodo da implementare: **doWork()**
    - Eseguito da un thread non-UI, gestito dal WorkManager

```kotlin
class MyWorker(c: Context, p: WorkerParameters)
    : Worker(c, p) {

  override fun doWork(): Result {
    // codice del task
    return Result.success()
  }
}
```

```java
public class MyWorker extends Worker {

  public MyWorker(Context c, WorkerParameters p) {
    super(c, p);
  }

  @Override
  public Result doWork() {
    // codice del task
    return Result.success()
  }
}
```

WorkRequest

- La richiesta di esecuzione di un task (ovvero, di un Worker) è definita da un'istanza di una sottoclasse di **WorkRequest**
- Il sistema fornisce due sottoclassi predefinite:
  - OneTimeWorkRequest – esegui il task una volta sola
  - PeriodicWorkRequest – esegui il task ripetutamente

```
val myWReq = OneTimeWorkRequestBuilder<MyWorker>().build()
```

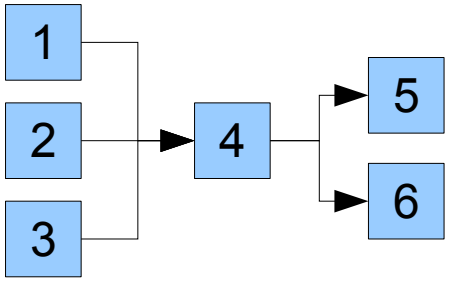
```
OneTimeWorkRequest myWReq = new OneTimeWorkRequest.Builder(MyWorker.class).build();
```

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

Struttura

- È possibile accodare più richieste insieme, che vengono svolte in parallelo, oppure metterle in sequenza

```
WorkManager.getInstance()
        .beginWith(wReq1, wReq2, wReq3)
        .then(wReq4)
        .then(wReq5, wReq6)
        .enqueue()
```



- È possibile annullare l'esecuzione di un task

```
WorkManager.cancelWorkById(wReq4.getId());
```

- … e altri metodi per verificare lo stato, ecc.

Vincoli

- Per finire, è possibile specificare quali vincoli (di sistema) devono essere verificati affinché un lavoro sia eseguibile
- Vincoli rappresentati come istanze di **Constraints**

```
val constr = Constraints.Builder()
    .setRequiresCharging(true)
    .setRequiresStorageNotLow(true)
    .build()

val wreq = OneTimeRequestBuilder<MyWReq>()
    .setConstraints(constr)
    .build()

WorkManager.getInstance().enqueue(wreq)
```

Vedere la documentazione delle varie classi per ulteriori dettagli.