Lezione 17

Programmazione Android

- I Services
  - Definizione
  - Service started e IntentService
  - Service bound

I Services

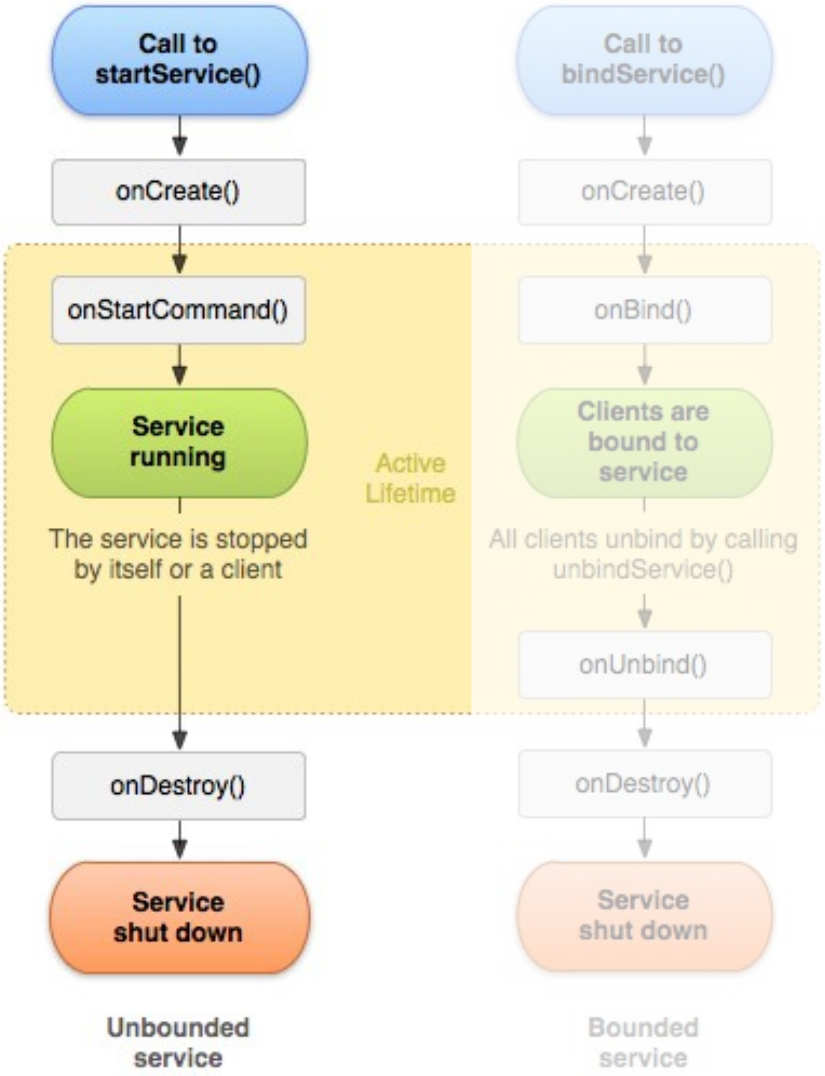
Service

- Dopo le Activity, i Content Provider e i Broadcast Receiver, i **Services** sono il quarto tipo di componente di un'applicazione
- Un Service è un oggetto che può eseguire codice *senza* disporre di un'interfaccia utente
  - In effetti, opera sempre “in background”
  - **Ma nel senso della UI, non dei processi / thread!**
    - **Il codice del service viene eseguito dal “main” thread (quello della UI)**
- Le Activity di un'applicazione possono avviare uno o più dei propri Service
  - Perché rimangano in esecuzione indefinitamente, o
  - Per effettuare un po' di lavoro, e poi terminare

Service

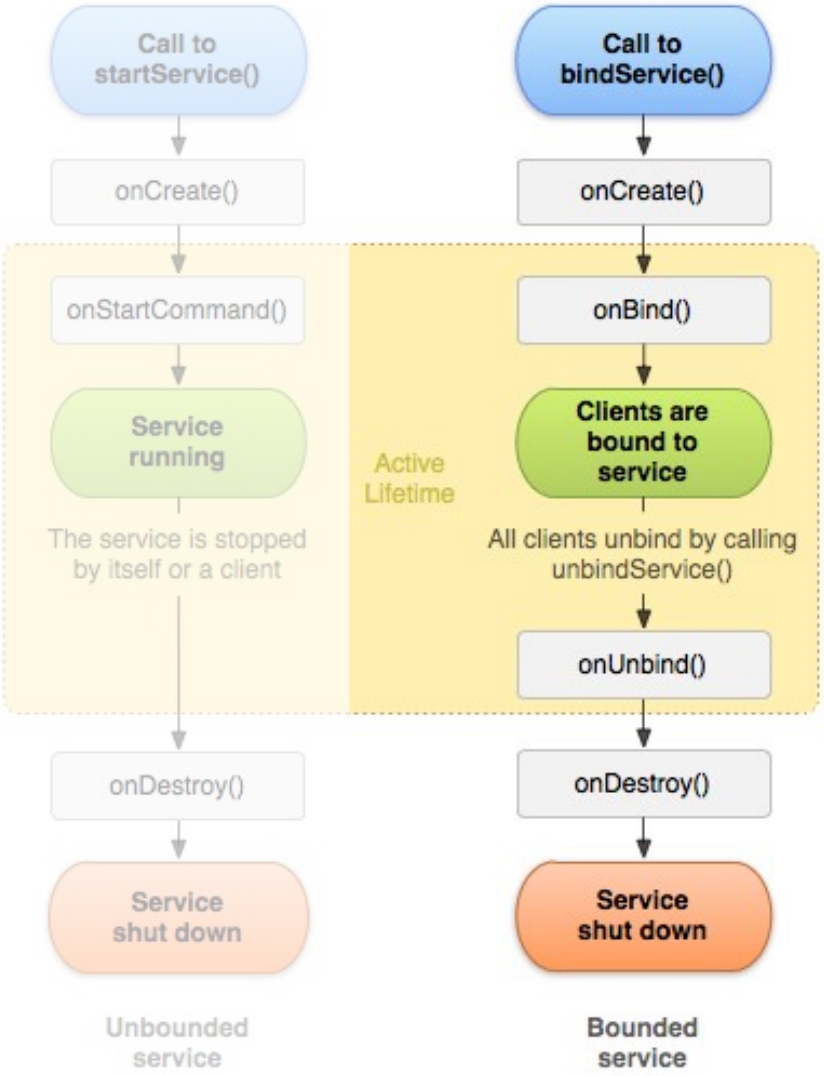
- Un Service ha un proprio ciclo di vita distinto
- Cambia a seconda che sia avviato per servire una singola richiesta (**started**), oppure per servire un flusso di richieste da un altro componente (**bound**)
- Il sistema può sempre uccidere un Service se ha necessità di memoria
  - Un Service bound ha sempre priorità almeno uguale a quella dell'activity (o altro componente) che ha chiesto un servizio
  - Generalmente, hanno priorità più alta delle Activity invisibili, e più bassa di quella delle Activity in primo piano

Ciclo di vita di un Service



- Nel caso di Service **started**, una call a startService() chiede che il service sia attivo
  - Se non era attivo, viene istanziato e inizializzato
  - Se era già attivo, non succede nulla
- Non esiste **nesting**

Ciclo di vita di un Service



- Nel caso di Service **bound**, il sistema tiene traccia di quanti clienti al momento usano il Service
- Quando il conteggio raggiunge 0, il Service può essere dismesso
- Le connessioni sono permanenti

Ciclo di vita di un Service

- Started
  - Il servizio richiesto ha un inizio
- Bound
  - Il servizio richiesto ha un inizio e una fine

startService()

bindService() unbindService()

Avviare un Service (started)

- Per avviare un Service, basta inviargli un Intent
  - Esplicito (alla classe) o implicito (ACTION ecc.)
- Esempio (da un'Activity):

```
Intent i = new Intent(this, TestService.class);
startService(i);
```

- Il sistema istanzierà il Service e chiamerà la sua onCreate() (se necessario)
- Poi passerà l'Intent a onStartCommand()

Avviare un Service (started)

- Per avviare un Service, basta inviargli un Intent
  - Esplicito (alla classe) o implicito (ACTION ecc.)
- Esempio (da un'Activity):

```
Intent i = new Intent(t
startService(i);
```

- Il sistema istanzierà il Ser
  onCreate() (se necessario)
- Poi passerà l'Intent a onStartCommand()

Rischio sicurezza: non sapete **quale** Service risponderà all'Intent!

Service minimale

```
public class TestService extends Service {

    @Override
    public void onCreate() {
        // Servizio appena creato
        super.onCreate();
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        // Servizio riceve un Intent
        return super.onStartCommand(intent, flags, startId);
    }

    @Override
    public IBinder onBind(Intent arg0) {
        // Servizio riceve un bind
        return null;
    }
}
```

Attenzione: questi metodi vengono chiamati sul thread della UI! Di solito, vengono subito creati dei Thread per fare il “lavoro vero”, e si torna immediatamente al chiamante.

- **intent** può essere null se è un restart
- **flags** è 0, START_FLAG_REDELIVERY (nuova consegna di un vecchio Intent) o START_FLAG_RETRY(nuovo tentativo di consegna dopo un fallimento)
- **startID** è un id numerico univoco della richiesta

Esempio: Musica!

(dal sample Random Music Player)

- Immaginiamo di voler scrivere un player musicale
- Il player avrà un'Activity con la sua interfaccia utente
- Il player dovrà ovviamente riprodurre i brani musicali
  - Sia quando la GUI è in primo piano
  - Sia quando la GUI è coperta da altro, o sono andato nella Home, o proprio uscito
    - Ovviamente, se rientro nella GUI devo poter controllare il playback!

AndroidManifest.xml

```
<activity android:name=".MainActivity"
   android:label="@string/app_title"
   android:theme="@android:style/Theme.Black.NoTitleBar">
   <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
   </intent-filter>
</activity>

<service android:exported="false" android:name=".MusicService">
   <intent-filter>
      <action android:name="com.example.android.musicplayer.action.TOGGLE_PLAYBACK" />
      <action android:name="com.example.android.musicplayer.action.PLAY" />
      <action android:name="com.example.android.musicplayer.action.PAUSE" />
      <action android:name="com.example.android.musicplayer.action.SKIP" />
      <action android:name="com.example.android.musicplayer.action.REWIND" />
      <action android:name="com.example.android.musicplayer.action.STOP" />
   </intent-filter>
   <intent-filter>
      <action android:name="com.example.android.musicplayer.action.URL" />
      <data android:scheme="http" />
   </intent-filter>
</service>
```

MainActivity.java

```
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        mPlayButton = (Button) findViewById(R.id.playbutton);
        /* ... */
        mEjectButton = (Button) findViewById(R.id.ejectbutton);

        mPlayButton.setOnClickListener(this);
        /* ... */
        mEjectButton.setOnClickListener(this);
}
```

```
public void onClick(View target) {
    if (target == mPlayButton)
        startService(new Intent(MusicService.ACTION_PLAY));
    else if (target == mPauseButton)
        startService(new Intent(MusicService.ACTION_PAUSE));
    else if (target == mSkipButton)
        startService(new Intent(MusicService.ACTION_SKIP));
    else if (target == mRewindButton)
        startService(new Intent(MusicService.ACTION_REWIND));
    else if (target == mStopButton)
        startService(new Intent(MusicService.ACTION_STOP));
    else if (target == mEjectButton) {
        showUrlDialog();
    }
```

MusicService.java

```
@Override
public void onCreate() {

    mNotificationManager = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
    mAudioManager = (AudioManager) getSystemService(AUDIO_SERVICE);

    // Create the retriever and start an asynchronous task that will prepare it.
    mRetriever = new MusicRetriever(getContentResolver());
    (new PrepareMusicRetrieverTask(mRetriever,this)).execute();

    mDummyAlbumArt = BitmapFactory.decodeResource(getResources(), R.drawable.dummy_album_art);
}

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    String action = intent.getAction();
    if (action.equals(ACTION_TOGGLE_PLAYBACK)) processTogglePlaybackRequest();
    else if (action.equals(ACTION_PLAY)) processPlayRequest();
    else if (action.equals(ACTION_PAUSE)) processPauseRequest();
    /* ... */
    else if (action.equals(ACTION_URL)) processAddRequest(intent);

    return START_NOT_STICKY; // Means we started the service, but don't want it to restart in case it's killed.
}
```

MusicService.java

- A partire dalle varie process...(), abbiamo la logica di gestione delle playlist e simili
  - Fino ad arrivare al play vero e proprio:

```
Url u = intent.getData().toString();
mPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);
mPlayer.setDataSource(u);
mPlayer.setVolume(1.0f, 1.0fl);
mPlayer.start();
```

- Il Service è in esecuzione anche se l'Activity è chiusa
  - Ma può sempre inviare un Intent per riattivarla, per esempio se la playlist è vuota
    - Meglio però farlo via Notification!

Terminare un service

- Una volta avviato con startService(), un servizio può essere terminato in tre modi
  - Il servizio stesso chiama stopSelf() – suicidio
    - stopSelf() – termina incondizionatamente
    - stopSelf(*id*) – segnala che è terminato il servizio della chiamata a onStartCommand(intent, flag, *id*) con l'*id* dato
    - stopSelfResult(*id*) – come sopra
  - Un altro componente chiama stopService(Intent i) passando un Intent che identifica il servizio – omicidio
  - Il sistema uccide forzosamente il servizio perché ha bisogno di memoria – genocidio

Terminare un service

- Una volta avviato con
essere terminato in tr
  - Il servizio stesso chia
    - stopSelf() – termina in
    - stopSelf(*id*) – segnala che è terminato il servizio della chiamata a onStartCommand(intent, flag, *id*) con l'*id* dato
    - stopSelfResult(*id*) – come sopra
  - Un altro componente chiama stopService(Intent i) passando un Intent che identifica il servizio – omicidio
  - Il sistema uccide forzosamente il servizio perché ha bisogno di memoria – genocidio

Le richieste di servizio **devono** essere terminate nello stesso ordine in cui sono state ricevute.

Se si chiama stopSelf(*id*) e *id* è la richiesta più recente, il servizio viene fermato anche se ci sono richieste precedenti ancora in esecuzione.

Terminazione e riavvio

- **onStartService()** restituisce un valore numerico che indica come gestire i restart forzosi
  - **START_STICKY**: se il servizio è stato fermato, appena possibile verrà chiamato nuovamente onStartService() per riavviarlo, passando un Intent null
    - Tipicamente usato con startService() / stopService()
  - **START_NOT_STICKY**: se il servizio è stato fermato, verrà riavviato solo se ci sono chiamate a onStartService(...,id) pendenti, ovvero non pareggiate da stopSelf(id)
    - Tipicamente usato con startService() / stopSelf(id)
  - **START_REDELIVERY_INTENT**: se il servizio è stato fermato, verrà riavviato passando a onStartService() l'Intent originale

Terminazione e riavvio

- L'argomento *flags* di onStartCommand() indica la ragione del riavvio
  - **0**: non è un riavvio (primo avvio)
  - **START_FLAG_REDELIVERY**: si tratta di una ri-consegna dell'Intent
    - Il sistema aveva ucciso il servizio prima che il processing fosse concluso, ossia prima della chiamata a stopSelf()
    - A seguito di START_REDELIVERY_INTENT da onStartCommand()
  - **START_FLAG_RETRY**: si tratta di un ri-avvio dopo uccisione forzosa
    - A seguito di START_STICKY da onStartCommand()

Ottenere un handle al Service

- Il metodo startService() restituisce un'istanza di **ComponentName**
- Questo oggetto rappresenta il componente Service
  - Può anche rappresentare un'Activity, un BroadcastReceiver o un ContentProvider: è una classe generale
- Se **startService()** restituisce null, l'Intent non è stato consegnato
- Altrimenti, si può usare il ComponentName per ottenere informazioni sul servizio
  - Es: **getClass()**, **getClassName()**, **getPackageName()**

Riassunto (Service started)

- Il componente che necessita di un Service prepara un Intent che lo identifichi
  - E mette dentro, come data o extra, tutto il necessario
- Il componente chiama **startService(intent)**
- Il sistema “trova” il Service giusto
  - Se necessario, lo lancia → **onCreate()**
  - Comunque, **onStartCommand(intent, flag, id)**
- Il Service serve l'intent
  - Eventualmente, invia una risposta al componente chiamante
    - Perché lo conosce, o via un PendingIntent passato come extra
- Il Service chiama **stopSelf(id)**
  - Oppure rimane indefinitamente in giro, finché qualcuno non chiama **stopService()**

La classe IntentService

- Quasi sempre, è utile che un Service usi uno o più thread separati per servire le richieste
  - Ricordate: i metodi del ciclo di vita di un Service sono eseguiti dal thread UI (come per gli altri componenti)
- **IntentService** è una sottoclasse di Service che:
  - Serve le richieste in un thread separato
  - In caso di richieste multiple, gestisce una coda
    - Nota: **non** gestisce richieste in parallelo; se vi serve il parallelismo, dovrete implementare voi il “giusto” meccanismo

IntentService

- IntentService crea un thread, looper, message queue, handler nella sua onCreate()
  - Dai sorgenti di Android:

```
@Override
public void onCreate() {
    // TODO: It would be nice to have an option to hold a partial wakelock
    // during processing, and to have a static startService(Context, Intent)
    // method that would launch the service & hand off a wakelock.
    super.onCreate();
    HandlerThread thread = new HandlerThread("IntentService[" + mName + "]");
    thread.start();
    mServiceLooper = thread.getLooper();
    mServiceHandler = new ServiceHandler(mServiceLooper);
}
```

IntentService

- Il ServiceHandler si limita a passare i messaggi (prelevati dalla coda) a un metodo **onHandleIntent()** implementato da IntentService

```
private final class ServiceHandler extends Handler {
    public ServiceHandler(Looper looper) {
        super(looper);
    }
    @Override
    public void handleMessage(Message msg) {
        onHandleIntent((Intent)msg.obj);
        stopSelf(msg.arg1);
    }
}
```

IntentService

- onHandleIntent() è un metodo astratto che voi dovete implementare in una vostra sottoclasse di IntentService

```
protected abstract void onHandleIntent(Intent intent);
```

- In definitiva:
  - Create una sottoclasse di IntentService
  - Fate overload di onHandleIntent()
  - Il vostro codice sarà eseguito da un thread separato
  - Le richieste vengono serializzate (non serve synchronized)
  - Quando la coda è vuota, il service (si) termina

Ottimizzazioni batteria

- A partire da Oreo (8.0+, target 26+), Android forza alcune “ottimizzazioni di batteria” per i service started:
  - Se un service è in qualche modo visibile all’utente, per esempio perché ha emesso una notifica, è considerato in *foreground* (nel senso UI)
  - In caso contrario, è considerato in *background*
    - Se l’app a cui il service appartiene è in foreground, tutto bene
    - Se l’app a cui il service appartiene è a sua volta in background, il service **non viene eseguito**
      - **Pur non essendo logicamente terminato**

Ottimizzazioni batteria

- Sempre da Android 8.0+, ci sono alcune nuove strutture per gestire queste limitazioni:
  - **JobIntentService** – come IntentService, ma implementata con Job periodici anziché con Service
  - **startForegroundService()** – come startService(), ma
    - consente a un'app in background di lanciare un service in foreground
    - Il service **deve** chiamare startForeground(id, Notification) per postare una notifica entro 5 secondi dall'avvio
      - Altrimenti, l'app viene uccisa. Tiè.

Considerate anche se usare **WorkManager** (che può eseguire job come Service) anziché i Service direttamente

Service bound

- I service started hanno un'interazione limitata con i loro utenti
- È possibile in alternativa effettuare un **binding**
  - Il service e il componente che lo usa vengono legati in modo più stabile e continuativo
  - La connessione fra i due rimane finché non viene fatto esplicitamente l'unbound
- Il componente può **chiamare** direttamente **metodi** del Service
  - In-process, con un **IBinder**
    - Interfaccia che definisce i metodi del Service chiamabili dall'esterno
  - Cross-process, con **AIDL**
    - Struttura analoga a CORBA, RMI, RPC ecc. – non lo vediamo

Iniziare un binding

- Il servizio **non** viene lanciato con startService(), ma con **bindService(intent, connection, flags)**
  - **intent**: l'Intent che identifica il Service, come prima
  - **connection**: un oggetto di classe ServiceConnection che controlla il tempo di vita del binding
    - onServiceConnected(ComponentName n, IBinder binder)
    - onServiceDisconnected(ComponentName n)
  - **flags**: precisa la gestione della priorità del servizio, per esempio BIND_IMPORTANT o BIND_NOT_FOREGROUND; BIND_AUTO_CREATE è un buon default

Usare un servizio bound

- La **bindService(intent, conn, flag)** causa una chiamata alla **onBind(intent)** del Service (e forse una **onCreate()**)
  - **bindService()** è void e termina subito: il binding poi è asincrono
- La **onBind(intent)** restituisce un nostro oggetto *binder* che implementa l'interfaccia **IBinder**
  - Spesso è una sottoclasse di **Binder**, e fornisce un metodo getter per il Service stesso
  - In teoria, può implementare un'interfaccia "pubblica" separata per il nostro Service
- Il *binder* viene passato alla **onServiceConnected()** di *conn*
- Da qui in avanti, il chiamante usa i metodi del *binder*
- Alla fine, si chiama **unbindService(*conn*)**

Usare un servizio bound

- Anche qui, abbiamo un rischio sicurezza se usiamo un Intent implicito in bindService()
  - Non possiamo sapere quale Service risponderà
  - Ma qui è peggio rispetto a prima
    - Perché poi ci aspettiamo di invocare dei metodi che magari il service che ha risposto non ha!
- Da Android 5.0+, non si può chiamare bindService() con un Intent implicito
  - Viene lanciata un'eccezione

Uno schema tipico (Activity)

```
public class LocalServiceBinding extends Activity {

   private LocalService mBoundService;

   private ServiceConnection mConnection = new ServiceConnection() {
      public void onServiceConnected(ComponentName className, IBinder service) {
         mBoundService = ((LocalService.LocalBinder)service).getService();
      }

      public void onServiceDisconnected(ComponentName className) {
         mBoundService = null;
      }
   };

   ....

   // bind
   bindService(new Intent(LocalServiceBinding.this, LocalService.class),
      mConnection, Context.BIND_AUTO_CREATE);

  // uso
  mBoundService.metodo(...);

   // unbind
    unbindService(mConnection);
```

- L'Activity definisce una Service Connection che si limita a memorizzare l'IBinder in un campo
- Poiché siamo in-process, possiamo fare un cast, e dare direttamente il tipo del nostro Service
- Sull'oggetto LocalService possiamo poi chiamare i metodi a piacimento
  - Occhio al multi-threading!

Uno schema tipico (Service)

- Il nostro Service implementa **onBind()**…
- …che restituisce il nostro **LocalBinder**...
- … che ha un metodo **getService()**...
- … che restituisce l'oggetto **LocalService**
- Alla fine, il cliente ha un puntatore al **Service**!

```
public class LocalService extends Service {
    public class LocalBinder extends Binder {
        LocalService getService() {
            return LocalService.this;
        }
    }

    private final IBinder mBinder = new
LocalBinder();

    @Override
    public void onCreate() { /* ... */    }

    @Override
    public void onDestroy() { /* ... */   }

    @Override
    public IBinder onBind(Intent intent) {
        return mBinder;
    }
```

Mescolare bound e unbound

- È perfettamente possibile che un Service offra sia un'interfaccia unbound che una bound
  - onStartCommand() → interfaccia unbound
  - onBind() → interfaccia bound
- Tuttavia, il ciclo di vita si fa complicato assai
  - Come se già non fosse complicato di suo
- Meglio, in generale, scegliere uno stile e mantenerlo

Riassunto

- Activity: ho una UI
- Service: non ho una UI (ma posso avere notifiche)
  - Unbound: il servizio processa singole richieste
    - START_STICKY: il servizio può essere ucciso mentre processa una richiesta; in tal caso riattivalo appena possibile (perdendo la richiesta)
    - START_NOT_STICKY: il servizio può essere ucciso mentre processa una richiesta, in tal caso non riattivarlo fino alla prossima startService()
    - START_REDELIVER_INTENT: come START_STICKY, ma in più inoltra gli Intent delle richieste non ancora terminate
  - Bound: il servizio non può essere ucciso mentre è bound