Services: background process

Programmazione Mobile

A.A. 2021/22

M.O. Spata

Service

- Android offre la classe **Service** per creare componenti di applicazioni che consentono la gestione di operazioni e funzionalità che dovrebbero operare in modo invisibile e che di fatto non hanno un'interfaccia utente.
- Abbiamo già visto come Android assegni ai **Service** una priorità maggiore rispetto alle **Activity** inattive; ciò riduce il rischio che Android li uccida in caso il sistema richieda risorse. In alcuni casi si può incrementare tale priorità per renderla uguale a quella di Activity in foreground.
- Utilizzando gli oggetti Service, è possibile garantire che le applicazioni continuino a funzionare e rispondere ad eventi, anche quando non sono effettivamente attive.

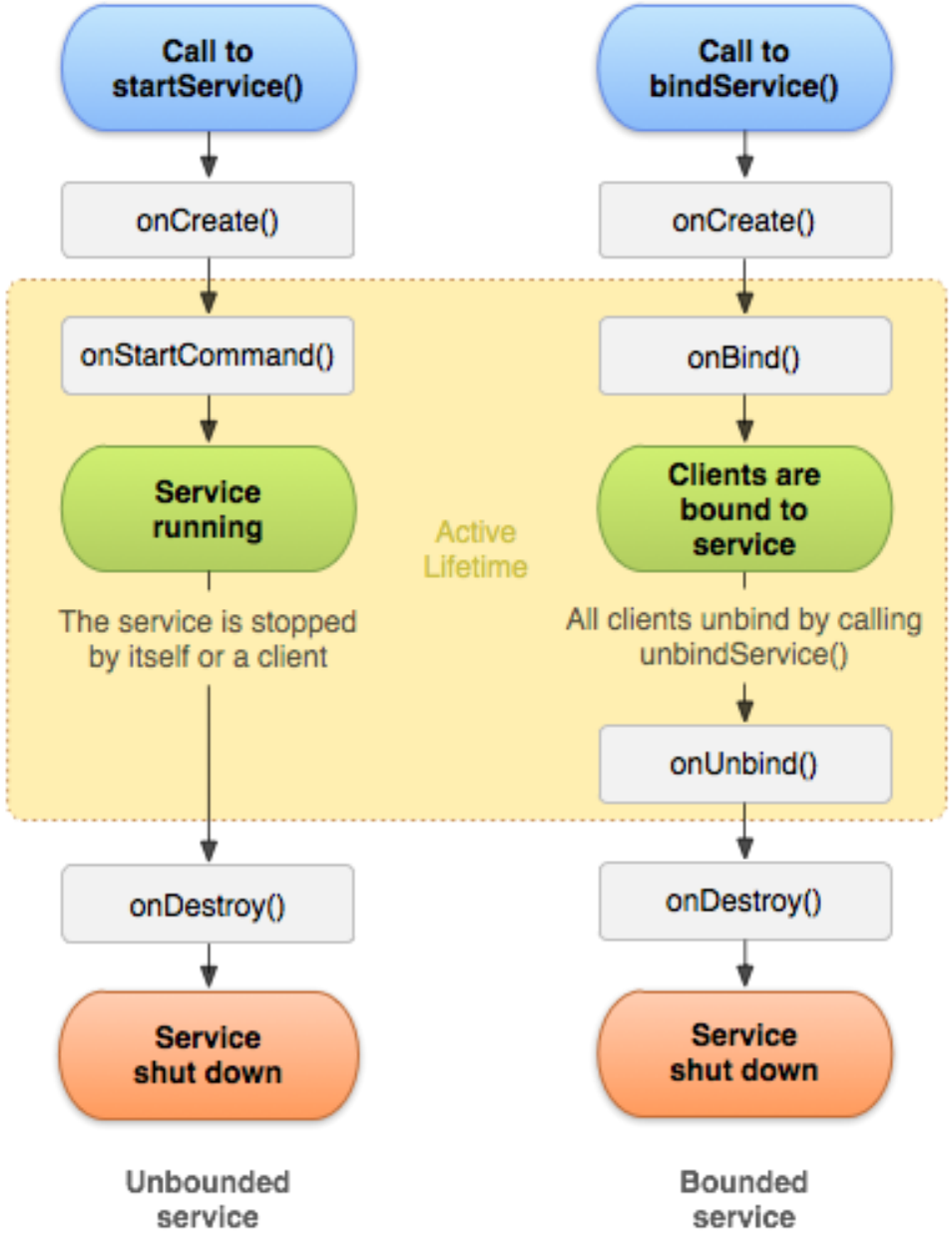
Toast e Notification

- Android offre diverse tecniche alle applicazioni per comunicare con gli utenti senza una determinata Activity. Ad esempio attraverso **Toast** e **Notification**.
- I **Toast** permettono attraverso diolog-box non-modali di visualizzare informazioni agli utenti senza togliere il focus alla applicazione attiva. Si tratta di messaggi in genere brevi, silenziosi e transitorie.
- Gli oggetti **Notification** rappresentano un meccanismo più solido per avvisare gli utenti. Di fatto gli eventi, comunicazioni ed informazioni vengono registrate e visualizzate attraverso icone nella barra di stato che opportunamente visualizzata ne mostra i dettagli.

Service

- A differenza delle Activity, che di solito presentano una interfaccia grafica molto ricca, i **Service** sono eseguiti in background e rappresentano la soluzione migliore per eseguire delle elaborazioni o per la gestione di eventi anche quando le Activity di una applicazione sono inattive o invisibili o addirittura sono state chiuse.
- I **Service** vengono avviati, stoppati e controllati attraverso componenti di altre applicazioni (Service, Activity, Broadcast Receivers) e sono da preferire a qualsiasi altro componente se si sta costruendo una applicazione la cui attività non dipende direttamente dall'input dell'utente.
- Android implementa nativamente diversi **Service** (Location Manager, Media Controller, Notification Manager).

Services


Call to startService()
onCreate()
onStartCommand()
Service running
The service is stopped by itself or a client
onDestroy()
Service shut down
Unbounded service
Active Lifetime
Call to bindService()
onCreate()
onBind()
Clients are bound to service
All clients unbind by calling unbindService()
onUnbind()
onDestroy()
Service shut down
Bounded service

Esempio

- Per definire un Service è necessario creare una classe che estende la classe **Service**.
- Di quest'ultima dovranno essere ridefiniti per lo meno i metodi: *onCreate* ed *onBind*.

```
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;

public class MyService extends Service {

  @Override
  public void onCreate() {
    // TODO: Actions to perform when service is created.
  }

  @Override
  public IBinder onBind(Intent intent) {
    // TODO: Replace with service binding implementation.
    return null;
  }
}
```

onStartCommand

- Inoltre nella maggior parte dei casi si dovrà ridefinire il metodo ***onStartCommand***; tale metodo è richiamato ogni qual volta viene fatto partire un servizio (ad esempio attraverso l'utilizzo della funzione ***startService***).
- Permette di specificare attraverso il valore di ritorno come gestire un eventuale **restart** nel caso in cui il servizio venisse interrotto o addirittura ucciso prima della sua fine regolare (chiamata alle funzioni *stopService* o *stopSelf*)

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
  // TODO Launch a background thread to do processing.
  return Service.START_STICKY;
}
```

Service

- I **Service** di per sè vengono lanciati nel thread dell'applicazione principale (come avviene per le activity) il che implica che ogni elaborazione fatta nel metodo onStartCommand di fatto ha ripercussioni sul thread principale.
- Per implementare un Service, così come noi lo abbiamo inteso ovvero per eseguire elaborazioni in background, sarà necessario creare ed avviare un nuovo **thread** all'interno della funzione *onStartCommand* e stoppare il Service quando l'elaborazione sarà completata.

Controllo di un Service: START_STICKY

- Il controllo della eventuale ripartenza del Service, in caso di terminazione anticipata del Service in questione, avviene controllando il valore di ritorno della funzione *onStartCommand*, un intero che può assumere diversi valori ovvero:
  - **START_STICKY**: Se viene restituito questo valore, la funzione **onStartCommand** verrà richiamata ogni qual volta il Service riparte dopo un stop imposto dal sistema a run-time. Normalmente si ritorna questo stato quando il Service gestisce il suo stato, ed esplicitamente, attraverso le chiamate alle funzioni **startService** e **stopService**, viene lanciato o stoppato.

Controllo di un Service: START_NOT_STICKY

- **START_NOT_STICKY**: Questo valore viene utilizzato per Service che sono stati avviati per eseguire specifiche azioni o comandi, e tipicamente terminano con la chiamata alla funzione stopSelf quando hanno completato le azioni loro richieste. A seguito della terminazione a run-time i servizi gestiti con tale valore di ritorno, verranno riavviati solo se esistono richieste di start in pending. Ovvero se nessuna chiamata alla funzione startService è stata fatta dopo che il Service è stato terminato il Service sarà stoppato senza richiamare la funzione onStartCommand.

Controllo di un service: START_DELIVERY_INTENT

- **START_DELIVERY_INTENT**: E' una combinazione delle due opzioni precedenti ritornando infatti questo valore nel caso in cui il Service sia stato terminato dal sistema a run-time, esso sarà fatto ripartire solo se ci sono chiamate di start pendenti o se il Service è stato interrotto prima della chiamata stopSelf. Tale modalità permette ed assicura che il comando richiesto al servizio sia effettivamente completato.

Controllo di un Service

- Il valore di ritorno di **onStartCommand** influenza i parametri della stessa funzione nella chiamata successiva.
- In particolare Inizialmente il servizio viene richiamato specificando alla funzione **startService** un Intent come parametro; questo intent lo ritroviamo come primo parametro della onStartCommand così come ritroveremo 0 nel parametro flag ed un eventuale id nell'ultimo parametro.

Controllo di un Service

- Dopo un restart dovuto al sistema il valore del parametro Intent potrà essere null (nel caso di START_STICKY) o avere il valore dell'intent iniziale (nel caso di START_DELIVERY_INTENT), mentre il parametro flag potrà avere uno dei seguenti valori:
  - **START_FLAG_DELIVERY**: indica che il parametro Intent è stato ridistribuito perchè il service è stato precedentemente interrotto dal sistema prima della chiamata esplicita a stopSelf.
  - **START_FLAG_RETRY**: Il Service è ripartito dopo una interruzione o terminazione inaspettata.

Esempio

- Chiaramente a partire dall'analisi di questi dati, e risalendo quindi alla causa del restart del Service possono essere fatte da codice operazioni diverse e legate al contesto specifico:

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
  if ((flags & START_FLAG_RETRY) == 0) {
    // TODO If it's a restart, do something.
  }
  else {
    // TODO Alternative background process.
  }
  return Service.START_STICKY;
}
```

Registrazione di un Service

- Una volta implementata la classe Service per il nuovo servizio da realizzare è necessaria la registrazione nell'application manifest attraverso il nodo <service> all'interno del nodo <application>

```
<service android:enabled="true" android:name=".MyService"/>
```

- E' inoltre possibile specificare attraverso l'attributo **requires-permission** quali **uses-permission** deve richiedere una applicazione per utilizzare il service in questione.

Avvio di Service: implicito o esplicito

- Per l'esecuzione di un Service è possibile utilizzare il metodo *startService*: esso riceve come parametro un Intent che pertanto consente un avvio implicito o esplicito:
  - *implicito*: si deve registrare un intent-filter con una action specifica per il service in questione, quindi usare l'azione per la creazione dell'intent
  - *esplicito*: creare l'intent specificando il nome della classe service

```
// Implicitly start a Service
Intent myIntent = new Intent(MyService.MY_ACTION);
myIntent.putExtra("TOPPING", "Margherita");
startService(myIntent);

// Explicitly start a Service
startService(new Intent(this, MyService.class));
```

Stop di un Service

- Per fermare un Service si utilizza il metodo *stopService* cui deve essere passato un intent che individua il Service da stoppare:

```
ComponentName service = startService(new Intent(this, BaseballWatch.class));
// Stop a service using the service name.
stopService(new Intent(this, service.getClass()));
// Stop a service explicitly.
try {
  Class serviceClass = Class.forName(service.getClassName());
  stopService(new Intent(this, serviceClass));
} catch (ClassNotFoundException e) {}
```

Stop di un Service

- Una volta che il Service ha completato una azione che era chiamato a fare può effettuare una chiamata al metodo ***stopSelf***, sia senza parametri per forzare uno stop, oppure specificando un parametro intero che rappresenta lo **startid** e che forza lo stop del Service solo dopo che tutte le elaborazioni sono state completate per tutte le istanze delle chiamate a ***startService***.

Priorità dei processi

- Abbiamo già visto come Android possa terminare i processi (Activities, Services) per una efficiente gestione delle risorse. Nell'assegnare la priorità ai processi e ai Service, viene assegnata una priorità inferiore solo a quella delle Activity in foreground.
- In tutti quei casi in cui il Service sta "interagendo" con l'utente (es. play su un brano musicale) sarebbe opportuno aumentare la sua priorità ovvero equipararlo ad un activity in foreground. Per fare ciò si utilizza il metodo ***startForeground***.

Foreground dei Service

- In genere l'utente è a conoscenza di tutte le applicazioni in foreground perchè c'è una interazione continua con esse.
- Ciò per i Service non è sempre vero, pertanto quando i Service sono riportati in foreground nella chiamata a **startForeground** si deve specificare un oggetto **Notification** che dovrà esistere fino a quando il Service sarà in foreground.

```
int NOTIFICATION_ID = 1;

Intent intent = new Intent(this, MyActivity.class);
PendingIntent pi = PendingIntent.getActivity(this, 1, intent, 0));
Notification notification = new Notification(R.drawable.icon,
  "Running in the Foreground", System.currentTimeMillis());
notification.setLatestEventInfo(this, "Title", "Text", pi);

notification.flags = notification.flags |
                     Notification.FLAG_ONGOING_EVENT;

startForeground(NOTIFICATION_ID, notification);
```

Processi in background

- L'utilizzo di processi in background è l'unico metodo utilizzabile per evitare il noiosissimo messaggio "Force Close" cha Android visualizza quando un Activity non risponde entro 5 sec ad un input utente o quando un Broadcast Receivers non completa l'esecuzione di onReceive entro 10 sec.
- In generale sarebbe opportuno utilizzare i **threads in background** per tutti i processi time-consuming quali operazioni su file, ricerca su rete, transazione su db e calcoli complessi.

Alternative per eseguire processi in background

- Android offre due alternative per permettere di eseguire in background i processi.
  - la classe **AsyncTask**: tale classe permette di definire una operazione da eseguire in background e fornire un gestore di eventi utilizzabile per monitorare lo stato di avanzamento dell'operazione ed inoltrare i risultati al thread della GUI.
  - i **Threads**: è possibile implementare la propria classe Thread ed utilizzare una classe Handler per la sincronizzazione con il Thread della GUI prima di aggiornare la UI.

AsyncTask

- La classe **AsyncTask** offre un semplice meccanismo per spostare le operazioni time-consuming in un thread in background.
- In particolare gestisce i thread sia nella fase di creazione, gestione e sincronizzazione dando la possibilità di creare processi asincroni consistenti nelle elaborazioni da fare in background e di aggiornare l'interfaccia utente quando l'elaborazione è stata completata.

AsyncTask

- L'implementazione della classe **AsyncTask** prevede che vengano specificati le classi utilizzate per i parametri della funzione *execute* che per l'appunto si riferiscono a : input, valori dello stato di avanzamento, risultati.

```
AsyncTask<[Input Parameter Type], [Progress Report Type], [Result Type]>
```

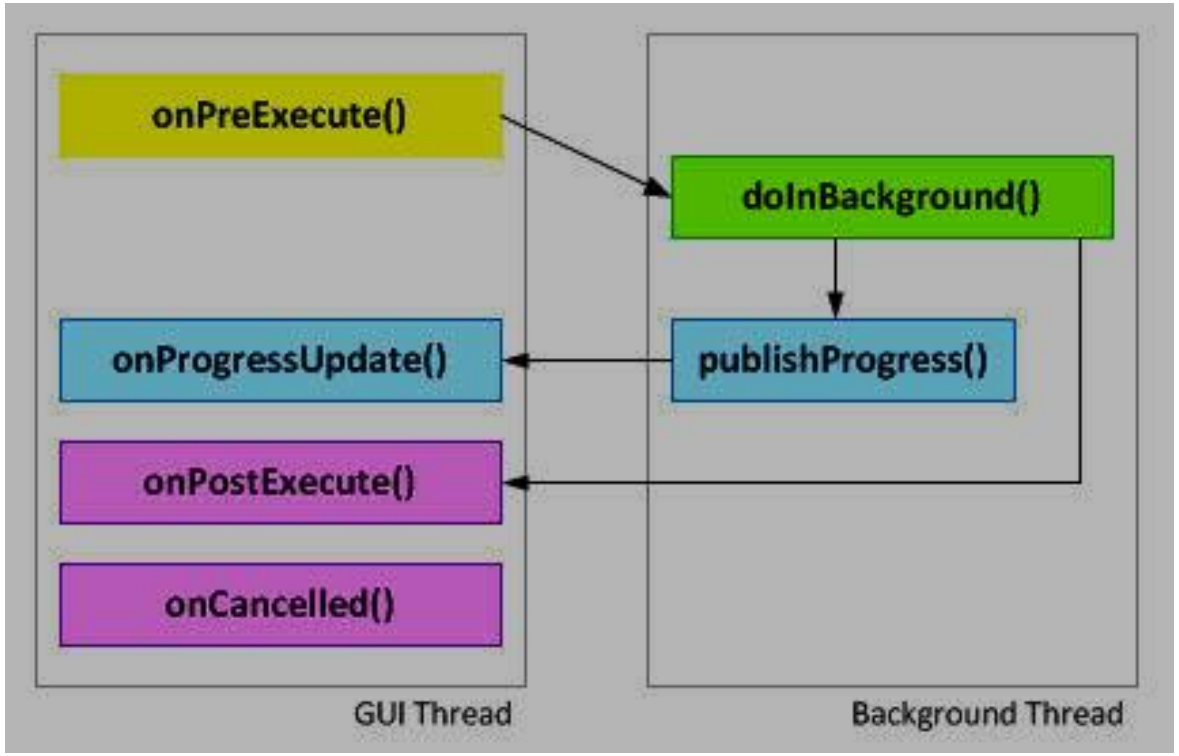
AsyncTask: esempio

- Per creare un nuovo task asincrono è necessario estendere la classe AsyncTask:

```
private class MyAsyncTask extends AsyncTask<String, Integer, Integer> {
  @Override
  protected void onProgressUpdate(Integer... progress) {
    // [... Update progress bar, Notification, or other UI element ...]
  }

  @Override
  protected void onPostExecute(Integer... result) {
    // [... Report results via UI update, Dialog, or notification ...]
  }

  @Override
  protected Integer doInBackground(String... parameter) {
    int myProgress = 0;

    // [... Perform background processing task, update myProgress ...]
    PublishProgress(myProgress)
    // [... Continue performing background processing task ...]

    // Return the value to be passed to onPostExecute
    return result;
  }
}
```

Metodi di AsyncTask

- La nuova classe deve reimplementare i seguenti metodi:
  - **doInBackground**: prende un insieme di parametri del tipo specificato nella implementazione della classe ed è il metodo che di fatto sarà eseguito nel thread in background, pertanto non deve tentare di interagire con oggetti dell'interfaccia utente. E' necessario inserire in questo metodo il codice time-consuming. Per la comunicazione dello stato di avanzamento è possibile utilizzare il metodo **publishProgress**.



Metodi di AsyncTask

- **onProgressUpdate**: permette di inoltrare aggiornamenti di stato intermedi al thread chiamante (UI): ovvero riceve l'insieme di parametri passati alla funzione publishProgress attraverso la chiamata nel metodo doInBackground. Si possono modificare elementi della UI.

- **onPostExecute**: riceve il valore di ritorno della funzione doInBackground; può essere usato per aggiornare la UI alla fine della esecuzione del task asincrono. Si possono modificare elementi della UI.

Metodi di AsyncTask

- Una volta implementata la nuova classe è possibile mandare in esecuzione il nuovo task asincrono creando una istanza della classe e richiamando il metodo **execute**.
- Ogni istanza può essere eseguita una sola volta.

```
new MyAsyncTask().execute("inputString1", "inputString2");
```

Thread

- L'alternativa all'utilizzo della classe **AsyncTask** è quella di creare nuovi **thread** proprietari e le relative classi per la gestione.
- In particolare si possono usare le classi **Handler** di Android per creare e gestire **thread** figli e la classe **java.lang.thread** per definire gli stessi thread.

Esempio di Thread

```
// This method is called on the main GUI thread.
private void mainProcessing() {
  // This moves the time consuming operation to a child thread.
  Thread thread = new Thread(null, doBackgroundThreadProcessing,
                             "Background");
  thread.start();
}

// Runnable that executes the background processing method.
private Runnable doBackgroundThreadProcessing = new Runnable() {
  public void run() {
    backgroundThreadProcessing();
  }
};

// Method which does some processing in the background.
private void backgroundThreadProcessing() {
  [ ... Time consuming operations ... ]
}
```

Esempio

- Per l'aggiornamento dell'interfaccia utente da un thread in background è possibile utilizzare la classe Handler come di seguito mostrato:

```
// Initialize a handler on the main thread.
private Handler handler = new Handler();

private void mainProcessing() {
  Thread thread = new Thread(null, doBackgroundThreadProcessing,
                             "Background");
  thread.start();
}

private Runnable doBackgroundThreadProcessing = new Runnable() {
  public void run() {
    backgroundThreadProcessing();
  }
};

// Method which does some processing in the background.
private void backgroundThreadProcessing() {
  [ ... Time consuming operations ... ]
  handler.post(doUpdateGUI);
}

// Runnable that executes the update GUI method.
private Runnable doUpdateGUI = new Runnable() {
  public void run() {
    updateGUI();
  }
};

private void updateGUI() {
  [ ... Open a dialog or modify a GUI element ... ]
}
```