Lezione 14-15

Programmazione Android

- Broadcast Receiver
- Esecuzione in background
  - Tematiche di threading
  - Alarm

Broadcast Receiver

Broadcast Receiver

- Dopo tutto quello che abbiamo visto su Activity, ContentProvider e Service, i Broadcast Receiver hanno poco di sorprendente
- 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
android:description	Descriptive text for the associated data.
android:enabled	Specify whether the receiver is enabled or not (that is, can be instantiated by the system).
android:exported	Flag indicating whether the given application component is available to other applications.
android:icon	A Drawable resource providing a graphical representation of its associated item.
android:label	A user-legible name for the given item.
android:logo	A Drawable resource providing an extended graphical logo for its associated item.
android:name	Required name of the class implementing the receiver, deriving from BroadcastReceiver.
android:permission	Specify a permission that a client is required to have in order to use the associated object.
android:process	Specify a specific process that the associated code is to run in.

Ciclo di vita di un BroadcastReceiver

- 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, o lanciare task asincroni senza risultato

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()

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

```
<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);
      }
    }
  }
}
```

Registrazione dinamica di un BroadcastReceiver

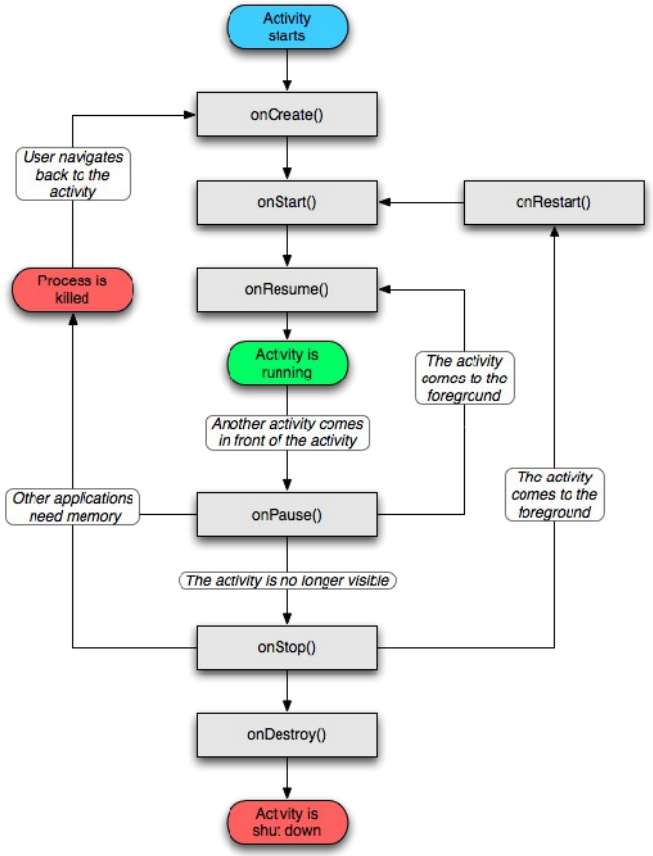
- È anche possibile registrare e deregistrare dinamicamente un BroadcastReceiver
  - In questo modo, l'applicazione riceve broadcast solo quando il receiver è registrato
- **public abstract** Intent registerReceiver (BroadcastReceiver receiver, IntentFilter filter)
- **public abstract** void unregisterReceiver (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 restituire dei risultati
  - Si invia il broadcast con il metodo
    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)

Multithreading

Ripasso sul threading

- **Staticamente**, un pezzo di codice appartiene
  - a un metodo, che appartiene
  - a una classe, che appartiene
  - a un package, che appartiene
  - a una applicazione
- **Dinamicamente**, un pezzo di codice è eseguito
  - da un thread, che appartiene
  - a un processo, che appartiene
  - a una applicazione

Può essere un Context

È un Context

In realtà, con opportuni attributi in AndroidManifest.xml si può condividere un processo fra più applicazioni

Ripasso sul threading

- Processo =
  - spazio degli indirizzi isolato
  - owner, diritti, eseguibile
  - stato (= contenuto della memoria)
- Thread =
  - flusso di esecuzione
  - stack delle chiamate
- In ogni istante, 0 o più thread di un processo sono in esecuzione

Ripasso Thread in Java
(in 1 lucido)

```
Thread t = new Thread(new Runnable() {
    public void run() {
        /* codice del job da eseguire */
    }
});

t.start();
```

```
o.wait();          o.notify();
```

```
synchronized (o) {
    /* eseguito in mutua esclusione su o */
}
```

```
synchronized void m(int a) {
    /* eseguito in mutua esclusione su this */
}
```

- La classe **Thread** rappresenta il thread
  - **Non** il codice da eseguire!
- L'interfaccia **Runnable** rappresenta il codice da eseguire
  - **Non** il thread che lo esegue!

Thread & Runnable

- L'interfaccia Runnable rappresenta **un task**: qualcosa da fare
  - Un solo metodo: public void run()
  - È la versione Java di un puntatore a funzione
    - L'oggetto che implementa Runnable sostanzialmente coincide con il corpo del suo metodo run()
- La classe Thread rappresenta un flusso di esecuzione
  - Nel senso classico: un PC, uno stack, ecc.
  - La memoria è **condivisa** all'interno del processo

Thread & Runnable

- L'oggetto **Thread** *rappresenta* un **thread** della JVM (o di Dalvik!), ma non lo *è*
  - Così come un oggetto File non è un file su disco, o un oggetto Socket non è un socket TCP/IP
- Finché non viene avviato, un Thread è semplicemente un oggetto Java in memoria
  - L'avvio avviene chiamando il metodo start() del Thread
  - Il metodo start() ritorna immediatamente al chiamante
  - Un nuovo thread parte l'esecuzione dal metodo run() del Thread

Thread & Runnable

- Primo metodo per lanciare un thread

```
class MioThread extends Thread {
   public void run() {
      /* codice da eseguire
         nel nuovo thread */
   }
}

…

Thread t = new MioThread();
t.start();
```

- Questo approccio lega strettamente il *thread* e il *task*
- In effetti, “sono” lo stesso oggetto!
- Né il thread né il task sono riutilizzabili

Thread & Runnable

- Secondo metodo per lanciare un thread

```
class MioTask
implements Runnable {
   public void run() {
      /* codice da eseguire
         nel nuovo thread */
   }
}
…
Runnable r = new MioTask();
Thread t = new Thread(r);
t.start();
```

- Questo approccio separa il *thread* e il *task*
- Sono due oggetti distinti
  - Il Runnable può anche essere una anonymous inner class

Sviluppo Applicazioni Mobili
Vincenzo Gervasi – a.a. 2012/13

Controllo di thread

- La classe Thread mette a disposizione una serie di metodi per controllare l'esecuzione
  - Controllo: start(), yield(), sleep(), interrupt(), join(), …
  - Setter: setName(), setPriority(), ...
  - Getter: getName(), getPriority(), getState(), interrupted(), isAlive(), …
  - Altro: gruppi di thread, class loader, eccezioni non gestite, ecc.
  - **NON USARE**: stop(), resume(), suspend(), destroy()

Sincronizzazione

- La sincronizzazione tra thread avviene attraverso l'uso di **monitor**
- Ogni oggetto Java ha un monitor associato
  - o.wait() - sospende il thread chiamante finché
    - viene fatto o.notify() (sullo stesso oggetto o)
    - Viene chiamato interrupt() sul thread sospeso
  - o.notify() - notifica gli eventuali thread sospesi sul monitor di o che uno di essi può ripartire
    - o.notifyAll() risveglia tutti i thread sospesi

Sviluppo Applicazioni Mobili
Vincenzo Gervasi – a.a. 2012/13

Sincronizzazione

- Prima di poter invocare o.wait() o o.notify(), un thread deve **acquisire il monitor** di o
- Questo può essere fatto tramite **synchronized**
  - Fornisce anche un semplice costrutto di mutua esclusione
  - Due varianti:
    - Statement: **synchronized** (*espr*) { *blocco* }
    - Dichiarazione **synchronized** *tipo m*(*arg*) { *blocco* }

Sincronizzazione

- Statement synchronized
  - Prova ad acquisire il monitor dell'oggetto denotato dall'espressione
  - Si sospende se il monitor è occupato
  - Rilascia il monitor all'uscita dal blocco

```
...
synchronized(expr)
{
    blocco
}
...
```

Sviluppo Applicazioni Mobili
Vincenzo Gervasi – a.a. 2012/13

Sincronizzazione

- Dichiarazione synchronized
  - Prova ad acquisire il monitor dell'oggetto (/classe) a cui appartiene il metodo di istanza (/statico)

```
T synchronized m(...) {
    corpo
}
static T synchronized m() {
    corpo
}
```

Sviluppo Applicazioni Mobili
Vincenzo Gervasi – a.a. 2012/13

Sincronizzazione

- I costrutti **synchronized** offrono un modo per realizzare la *mutua esclusione* e per *serializzare l'accesso* da parte di diversi thread
  - Particolare cura va posta nel proteggere le strutture dati condivise fra più thread!
  - Si possono usare le varianti “protette” delle collezioni
- I monitor acquisiti vengono rilasciati quando un thread si sospende (es., o.wait()) e riacquisiti al risveglio (es., o.notify())
  - L'I/O di sistema incorpora wait e notify sulle operazioni lunghe

Sistema e callback

- Come abbiamo visto in numerosissimi casi, le applicazioni si limitano a definire dei metodi callback
  - Ciclo di vita dell'Activity: onCreate(), onPause(), ...
  - Interazione con l'utente: onClick(), onKey(), onCreateOptionsMenu(), ...
  - Disegno della UI: onMeasure(), onDraw(), …
  - E tantissimi altri!
- Il thread di sistema che chiama questi metodi è detto **Thread della UI**

Le due regole auree

- **Mai usare il thread UI per operazioni lunghe**
- **Mai usare un thread diverso dal thread UI per aggiornare la UI**

- Problema
  - Come posso fare se serve una operazione lunga che deve aggiornare la UI?
    - Es.: accesso a DB, accesso alla rete, calcoli “pesanti”
  - Creare nuovi Thread mi aiuta per la regola #1, non per la #2

AsyncTask

- Il caso più comune è quando
  - Il thread UI deve far partire un task (lungo)
  - Il task deve aggiornare la UI durante lo svolgimento
  - Il task deve fornire il risultato alla UI alla fine
- Per questo particolare caso, è *molto* comodo usare la classe (astratta e generica) **AsyncTask**
  - Come in altri casi, dovremo creare una nostra sottoclasse e fare override di metodi

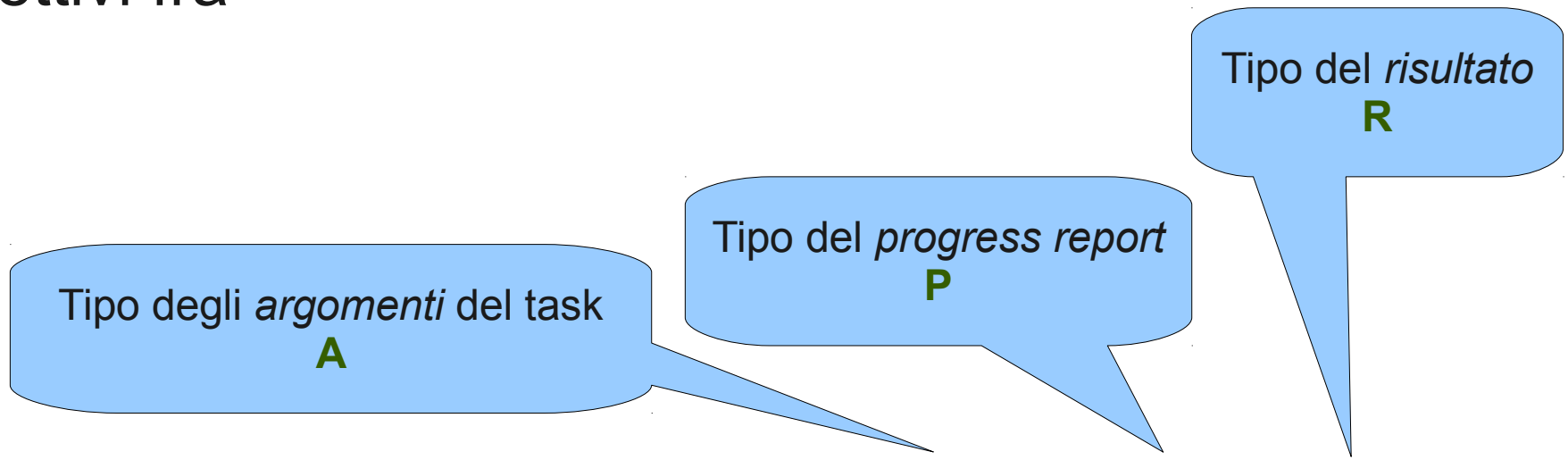
AsyncTask

```
class MyTask extends AsyncTask<Integer, Float, Void> {

  @Override
  protected Void doInBackground(Integer... params) {
    int limit=params[0], sleep = params[1];
    for (int i=0; i<limit && !isCancelled(); i++) {
      try {
        Thread.sleep(sleep);
      } catch (InterruptedException e) { ; }
      publishProgress((float)i/limit);
    }
    publishProgress(1.0f);
    return null;
  }

  @Override
  protected void onProgressUpdate(Float... p) {
    progressbar.setProgress((int) (p[0]*100));
  }
}
```

- Un task **deve** implementare doInBackground()
  - È un metodo astratto!
- Un task **può** implementare altri metodi
  - AsyncTask ne fornisce una implementazione vuota, esempio: onProgressUpdate()

AsyncTask

- AsyncTask è una **classe generica**
  - Può operare su tipi diversi
  - Al momento dell'istanziazione, si specificano i tipi effettivi fra < >




```
class MyTask extends AsyncTask<Integer, Float, Void>
```

AsyncTask

- Metodi da implementare
  - Ciclo naturale
    - void onPreExecute()
    - **R** doInBackground(**A**...)
    - void onProgressUpdate(**P**...)
    - void onPostExecute(**R**)
  - Cancellazione anticipata
    - void onCancelled(**R**)

Questo è l'uso tipico: ma nessuno vieta, per esempio, di chiamare getStatus() da un handler, o isCancelled() dall'esterno...

- Metodi da chiamare dall'esterno
  - Costruttori
  - AsyncTask execute(**A**...)
  - cancel(boolean interrupt)
  - **R** get()
  - AsyncTask.Status getStatus()
- Metodi da chiamare dagli on...()
  - void publishProgress(**P**...)
  - boolean isCancelled()

AsyncTask

- Metodi da implementare
  - Ciclo naturale
    - void onPreExecute()
    - **R** doInBackground(**A**...)
    - void onProgressUpdate(**P**...)
    - void onPostExecute(**R**)
  - Cancellazione anticipata
    - void onCancelled(**R**)

- Metodi da chiamare dall'esterno
  - Costruttori
  - AsyncTask execute(**P**...)
  - cancel(boolean interrupt)
  - **R** get()
  - AsyncTask.Status getStatus()
- Metodi da chiamare dagli on...()
  - void publishProgress(**P**...)
  - boolean isCancelled()

- **Metodi che sono eseguiti dal thread UI**
  - Devono essere veloci, ma possono interagire con la UI
- **Metodi che sono eseguiti dal thread in background**
  - Possono essere lenti, ma non devono interagire con la UI (o invocare altre funzioni del toolkit)

AsyncTask

- Esecuzione normale
  - Costruttore
  - execute(**A**...)
  - OnPreExecute()
  - **R** doInBackground(**A**...)
    - IsCancelled() → false
    - publishProgress(**P**...)
    - onProgressUpdate(**P**...)
    - …
  - onPostExecute(**R**)
  - **R** get() → risultato
- Esecuzione cancellata
  - Costruttore
  - execute(**A**...)
  - OnPreExecute()
  - **R** doInBackground(**A**...)
    - IsCancelled() → true (esce)
    - publishProgress(**P**...)
    - onProgressUpdate(**P**...)
    - …
  - onCancelled(**R**...)
  - **R** get() → CancelledException

Altri casi di esecuzione asincrona

- AsyncTask è solo una *classe di utilità* per organizzare i thread in uno schema frequente
- Ci sono comunque primitive per fare comunicare i thread non-UI con il thread UI in altre strutture
- In qualche caso, Android offre garanzie specifiche sul modello di threading che riducono la necessità di usare **synchronized**
  - **Nota bene**: se mai il thread UI dovesse incontrare un **synchronized**, sarebbe bloccato finché il thread che attualmente possiede il monitor non ha finito!

runOnUiThread()

- La classe Activity offre

  void runOnUiThread(Runnable r)

  Può essere chiamato da un thread non-UI
- Il runnable sarà eseguito dal thread UI dell'activity (in qualche momento del futuro)
  - Utile, per esempio, per
    - Aggiornamenti “volanti” di una progress bar
    - Rinfrescare una ListView man mano che arrivano dati
    - Fare un fade-in di immagini scaricate da rete

post()

- La classe View offre

void post(Runnable r)

void postDelayed(Runnable r, long millis)

- Possono essere chiamati da un thread non-UI
- Il runnable sarà eseguito dal thread UI dell'activity a cui questa View appartiene (dopo che siano trascorsi almeno *millis* ms)
- Non può essere invocato se la View non è inserita nel Layout di un'Activity!

post()

Thread non-UI

```
progress.post(new Runnable() {
    public void run() { progress.setProgress(k); }
} );
```

Thread UI

- Tipicamente, la post() viene invocata sulla View che deve essere manipolata
- Come al solito, si fa uso di *anonymous inner classes*
  - Ruolo analogo ai *delegate* di C# e ai *blocchi* di Objective-C
  - Ricordate che le *inner classes* hanno visibilità sulla chiusura lessicale del loro "contenitore"
    - Variabili locali dichiarate **final**
    - Variabili di istanza e di classe

Scavando scavando...

- Se classi e metodi di utilità messi a disposizione dalla libreria non bastano, si può scendere al livello sottostante
  - **Handler** – gestisce la MessageQueue di un thread
  - **Message** – busta per un Bundle
  - **MessageQueue** – coda di Message
  - **Looper** – classe che offre un ciclo lettura-dispatch da MessageQueue
- Ogni Activity ha un Looper eseguito dal thread UI
  - I vari post() accodano nella MessageQueue del Looper dell'Activity un Message con la specifica dell'operazione richiesta (come Parcelable)
- Siamo alle fondamenta di Android (package android.os.*)

Handler di utilità

- Android fornisce alcune classi di utilità per semplificare l'uso di handler
- Esempio: AsyncQueryHandler (per Content Provider)

```
class MyAQH extends AsyncQueryHandler {
  public MyAQH(ContentResolver cr) {
    super(cr);
  }

  @Override
  protected void onQueryComplete(int token, Object cookie,Cursor cursor) {
    /* ... */
  }
}
```

Struttura analoga per

startDelete() / onDeleteComplete()
startInsert() / onInsertComplete()
startUpdate() / onUpdateComplete()

Uso:

```
MyAQH asyncMusic = new MyAQH(getContentResolver());
asyncMusic.startQuery(token, cookie, uri, projection, selection, args, sort);
```

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 si 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()