Lezione 14

Programmazione Android

- Esecuzione concorrente
  - Tecniche per il multithreading
  - AsyncTask
  - Handler, Looper e le code messaggi

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, o di ART), 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

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

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
    - Comando: **synchronized** (*espr*) { *blocco* }
    - Dichiarazione: **synchronized** *tipo m*(*arg*) { *blocco* }

Sincronizzazione

- Comando **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
}
...
```

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
}
```

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

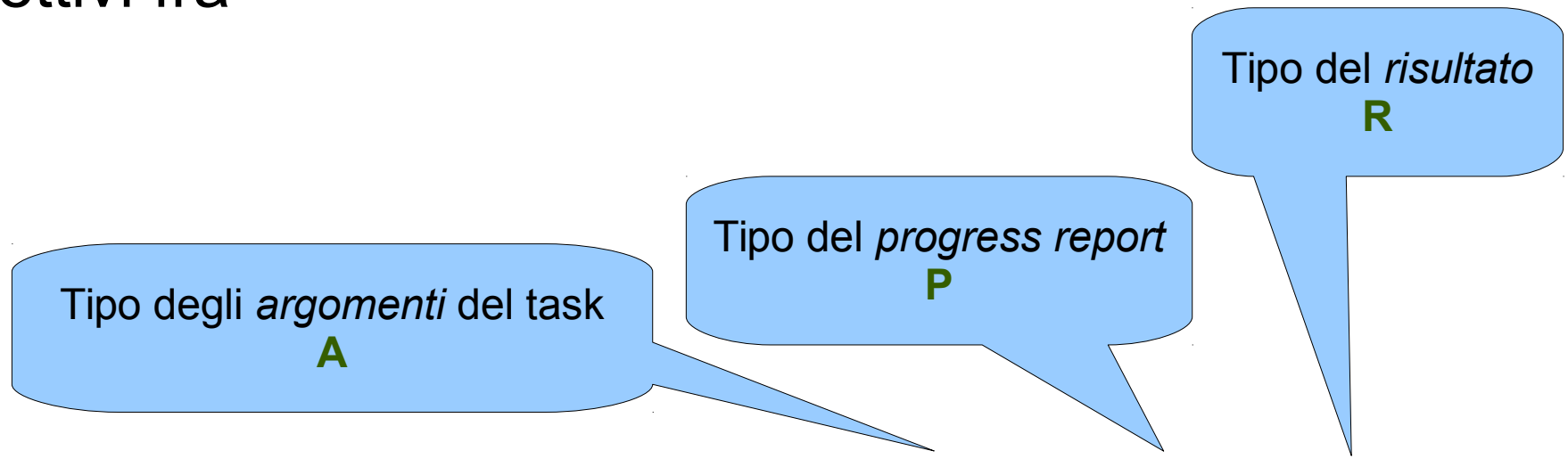
AsyncTask

```
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(**A**...)
  - 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#, ai *blocchi* di Objective-C, alle *chiusure* di Swift
  - Ricordate che le *inner classes* hanno visibilità sulla chiusura lessicale del loro "contenitore"
    - Variabili locali dichiarate **final**
    - Variabili di istanza e di classe

Esempio

```
public void onClick(View v) {
   new Thread(new Runnable() {
      public void run() {
         Bitmap b = caricaDaRete();
         iv.post(new Runnable() {
            public void run() {
               iv.setImageBitmap(b);
            }
         }
      }
   }).start();
}
```

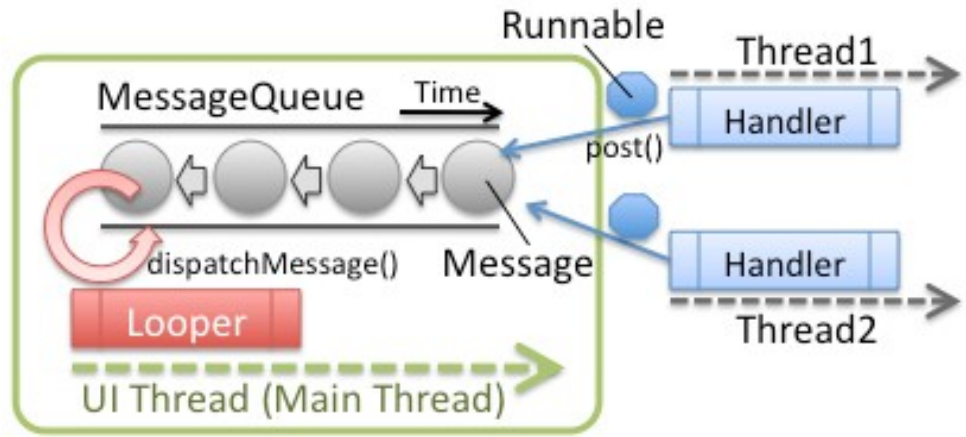
Esempio

```
public void onClick(View v) {                          Thread UI
   new Thread(new Runnable() {
      public void run() {                              Nuovo thread
         Bitmap b = caricaDaRete();
         iv.post(new Runnable() {
            public void run() {                        Thread UI
               iv.setImageBitmap(b);
            }
         }
      }
   }).start();
}
```

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.*)

Scavando scavando...



È possibile (ma non comune) creare la propria struttura di Handler, Looper ecc. e farla eseguire da un insieme di thread proprio, magari gestito da un ThreadPool configurato in maniera particolare.

Si tratta di usi avanzati che richiedono molta cautela!

- In effetti, tutte le volte che abbiamo detto:
  - "dopo la richiesta il sistema, con suo comodo, in qualche punto del futuro, farà la tale operazione"
- si intendeva:
  - la richiesta crea un Message che descrive l'operazione
  - lo passa all'Handler
  - che lo accoda nella MessageQueue
  - da cui verrà estratto da un Looper
  - che eseguirà l'operazione
- Esempio: invalidate()

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);
```