
# Lezione 15

# Programmazione Android

- Esecuzione concorrente: casi tipici
  - Download di immagini da web
  - Cursori asincroni
  - Download HTTP con il servizio DownloadManager
  - AsyncPlayer

# Download di immagini da web

# ImageDownloader

```
public class ImageDownloader {

  public void download(String url, ImageView imageView) {
    BitmapDownloaderTask task =
                              new BitmapDownloaderTask(imageView);
    task.execute(url);
  }
}
```

- Vogliamo una classe in grado di scaricare un'immagine dal web, e inserire la bitmap corrispondente in una ImageView
  - Scaricamento → su thread separato
  - Inserimento nella view → su thread UI

# BitmapDownloaderTask

- Ci viene in soccorso la AsyncTask!

```
class BitmapDownloaderTask extends AsyncTask<String, Void, Bitmap> {
    private String url;
    private final WeakReference<ImageView> imageViewReference;

    public BitmapDownloaderTask(ImageView imageView) {
        imageViewReference = new WeakReference<ImageView>(imageView);
    }

    @Override
    // Actual download method, run in the task thread
    protected Bitmap doInBackground(String... params) {
        // params comes from the execute() call: params[0] → url
        return downloadBitmap(params[0]); // returns bitmap
    }
```

segue

# BitmapDownloaderTask

```
    @Override
    // Once the image is downloaded, associates it to the imageView
    protected void onPostExecute(Bitmap bitmap) {
        if (isCancelled()) {
            bitmap = null;
        }

        if (imageViewReference != null) {
            ImageView imageView = imageViewReference.get();
            if (imageView != null) {
                imageView.setImageBitmap(bitmap);
            }
        }
    }
}
```

# Weak References

- Normalmente, tenere un riferimento (=puntatore) a un oggetto, impedisce al garbage collector di disallocarlo
  - Esempio:
    - BitmapDownloaderTask ha un riferimento alla ImageView
    - ImageView ha un riferimento a Context
    - Context ha riferimenti alla vita, all'universo e a tutto quanto
  - Risultato: anche se l'Activity è stata chiusa, la memoria occupata non può essere liberata
    - Almeno finché non termina il thread di scaricamento...

# Weak References

- Per evitare questa situazione, abbiamo usato una **WeakReference**
- Una weak reference si comporta come un riferimento, ma **non** impedisce al garbage collector di disallocare la memoria
  - Sempre che non ci siano altri riferimenti in giro!
- In caso di disallocazione, il metodo get() della WeakReference restituisce **null**

# (Download)

- La downloadBitmap(url) può essere realizzata in vari modi
  - Usando la classe **URL**:

```
Bitmap downloadBitmap(String url) {
    InputStream is = new URL(url).openStream();
    Bitmap b =  BitmapFactory.decodeStream(is);
    return b;
}
```

**Attenzione**: mancano controlli d'errore!

  - Usando la classe **AndroidHttpClient**
    - Offre un controllo completo sul trasferimento, ma un po' più complessa da usare
  - … e in numerosi altri modi

# In Kotlin

```
class ImageDownloader {
    fun download(url: String, imageView: ImageView) {
        val task = BitmapDownloaderTask(imageView)
        task.execute(url)
    }
}
```

- Notate nuovamente come la *type inference* riduce la necessità di attributi nelle dichiarazioni…
- Naturalmente, dobbiamo definire anche qui la **BitmapDownloaderTask**

# In Kotlin

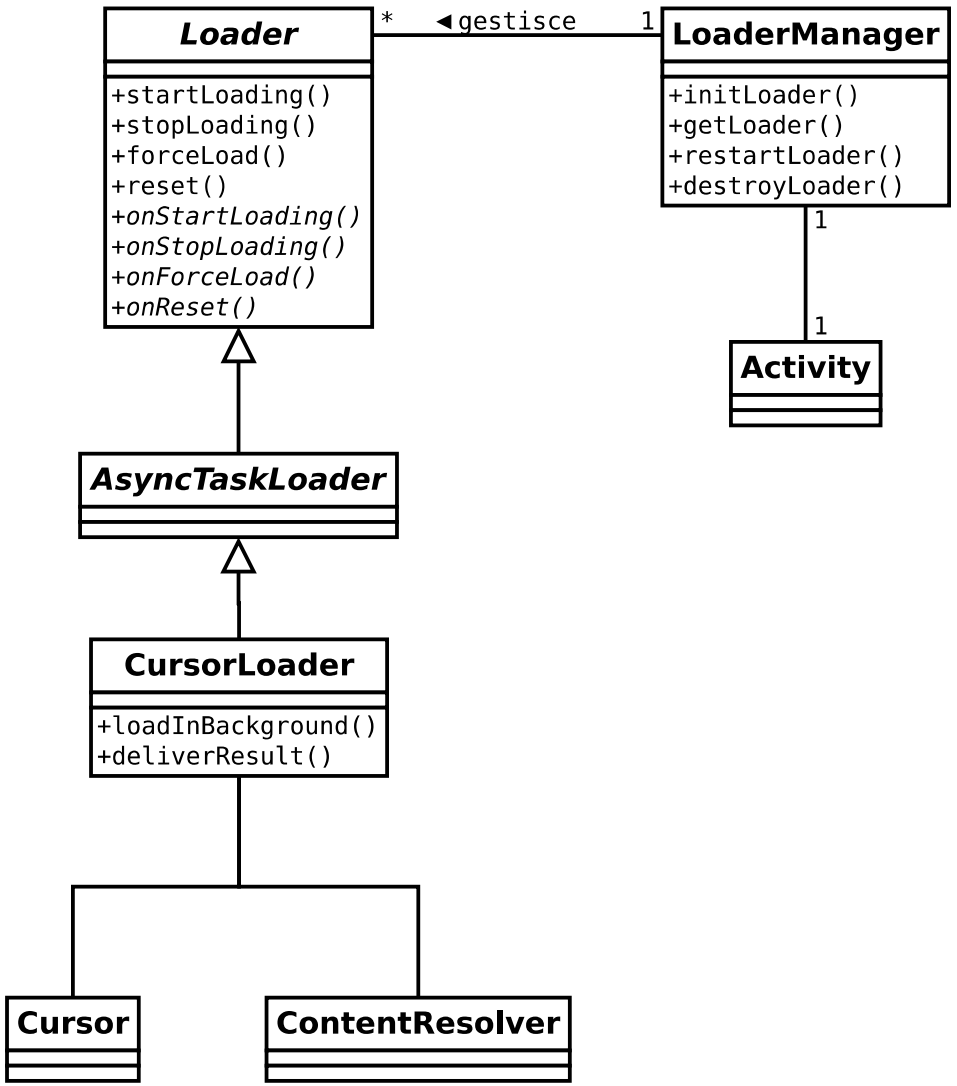
```
internal class BitmapDownloaderTask(imageView: ImageView) : AsyncTask<String, Void, Bitmap>() {
    private val url: String? = null
    private val imageViewReference: WeakReference<ImageView>?

    init { imageViewReference = WeakReference(imageView) }

    override fun doInBackground(vararg params: String): Bitmap? {
        return downloadBitmap(params[0])
    }

    override fun onPostExecute(bitmap: Bitmap?) {
        var bitmap = bitmap
        if (isCancelled) bitmap = null
        if (imageViewReference != null) {
            val imageView = imageViewReference.get()
            imageView?.setImageBitmap(bitmap)
        }
    }

    private fun downloadBitmap(url: String): Bitmap? {
        val `is` = URL(url).openStream()
        return BitmapFactory.decodeStream(`is`)
    }
}
```

# Cursori asincroni

# Loader

- La classe astratta **Loader** specifica un protocollo generico per il caricamento asincrono di dati
- Si tratta di componenti **attivi**
  - A loader attivo, quando i dati sottostanti vengono aggiornati, il Loader deve trasferire gli aggiornamenti al suo utente
  - Il trasferimento è **spesso** (ma non necessariamente) asincrono
  - Tipicamente associati ai Cursor

# Loader & co.




- Ogni Activity (o Fragment) ha associato un LoaderManager
  - Questo integra il ciclo di vita dei Loader in quello dell'Activity
- CursorLoader è una sottoclasse di Loader
  - Interroga un ContentResolver in background

# Inizializzare un Loader

- L'inizializzazione del LoaderManager viene tipicamente fatta nella onCreate() dell'Activity
- Si occupa anche di ripristinare lo stato (salvato) dei Loader

```
@Override
public void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    …
    getLoaderManager().initLoader(0, null, this);
    …
}
```

Un nostro oggetto che implementa LoaderCallbacks:
onCreateLoader()
onLoadFinished()
onLoaderReset()

ID del loader

Parametri del costruttore del loader

# Inizializzare un Loader

- L'ID numerico identifica univocamente un Loader
  - Se ne possono avere più di uno nella stessa Activity
- Comportamento tipico:
  - Se non esiste già un Loader con quell'ID, viene creato e inizializzato
    - Passando il secondo parametro al suo costruttore
  - Se invece esiste già, viene ripristinato il suo stato precedente (se c'è) dal Bundle
    - Per re-inizializzare un Loader già esistente: restartLoader()

# Avviare il caricamento

- Il nostro onCreateLoader() deve restituire un Loader inizializzato
- Per esempio, creando un CursorLoader:

```
public Loader<Cursor> onCreateLoader(int id, Bundle args)
{
    …
    return new CursorLoader(context, uri,
           projection, selection, args, sortOrder);
}
```

- Il CursorLoader fa partire la query (asincrona)

# Ricevere i dati

- Il nostro onLoadFinished() verrà chiamato quando la query (asincrona) avrà restituito un Cursor

```
public void onLoadFinished(Loader<Cursor> l, Cursor c) {
    // utilizza i dati nel Cursor
    // per esempio, per collegarli a un SimpleCursorAdapter:
    // adapter.swapCursor(c);
}
```

- **Nota**: il Cursor è di proprietà del LoaderManager
  - Provvede lui a gestirne il ciclo di vita

# Reset di un Loader

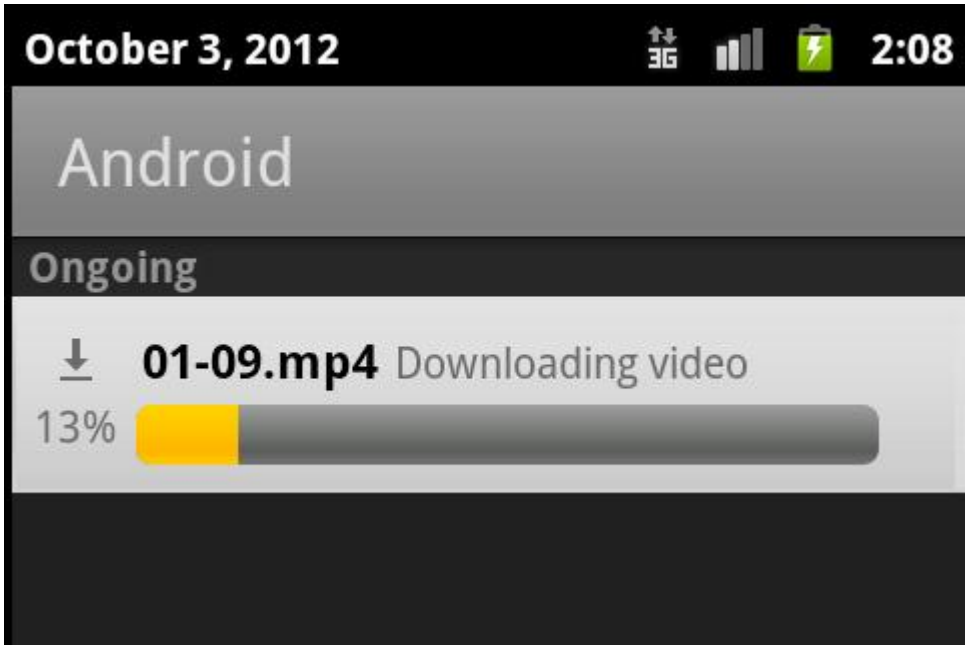
- Quando un Loader sta per essere chiuso (o ripristinato), il LoaderManager invoca onLoaderReset():

```
public void onLoaderReset(Loader<Cursor> l) {
    // L'ultimo cursore passato a onLoadFinished() sta per
    // essere chiuso. Dobbiamo smettere di usarlo. Per esempio,
    // se è stato usato un SimpleCursorAdapter, possiamo fare
    // adapter.swapCursor(null);
}
```

# Download HTTP con DownloadManager

# DownloadManager

- Android offre un servizio di sistema per il download di risorse via HTTP
- Il servizio è (ovviamente) asincrono
  - Gestisce lo scaricamento, la notifica di progresso, e segnala il completamento all'applicazione
  - Si occupa di restart se cade la connessione, ecc.



# Accedere al DownloadManager

- Come per tutti i servizi di sistema, si può usare la getSystemService()

```
DownloadManager dm = getSystemService(DOWNLOAD_SERVICE);
```

- Una volta ottenuto il servizio, si possono accodare download con

```
long id = dm.enqueue(request);
```

- E al termine, essere avvisati da un broadcast ACTION_DOWNLOAD_COMPLETE
  - L'app dovrà registrare un BroadcastReceiver

# Le richieste per il DownloadManager

- La classe DownloadManager.Request incapsula una richiesta di scaricamento
- Contiene tutti i dettagli del caso, fra cui
  - L'URI da scaricare (obbligatoria!)
  - Dove mettere il file locale (per default, spazio shared)
  - Se visualizzare una notifica o no durante il download
  - Un titolo e una descrizione (per le notifiche, se ci sono)
  - Restrizioni sulle reti da usare
    - Solo in wi-fi, anche in cellulare, anche in roaming, ecc.

# Esempio: avvio download

- Immaginiamo di avere un pulsante “download”...

```
public void onClick(View view) {
    dm = (DownloadManager)getSystemService(DOWNLOAD_SERVICE);
    u = Uri.parse(bella_url); // Uri u
    r = new Request(u); // Request r
    r.setTitle(“Una bella Url”);
    r.setDescription(“probabilmente fantastica!”);
    r.setAllowedOverRoaming(false); // non così fantastica!
    r.setVisibleInDownloadsUi(true);
    id = dm.equeue(r); // long id
}
```

# Esempio: completamento download

- Registriamo un receiver

```
registerReceiver(receiver,
   new IntentFilter(DownloadManager.ACTION_DOWNLOAD_COMPLETE)
);
```

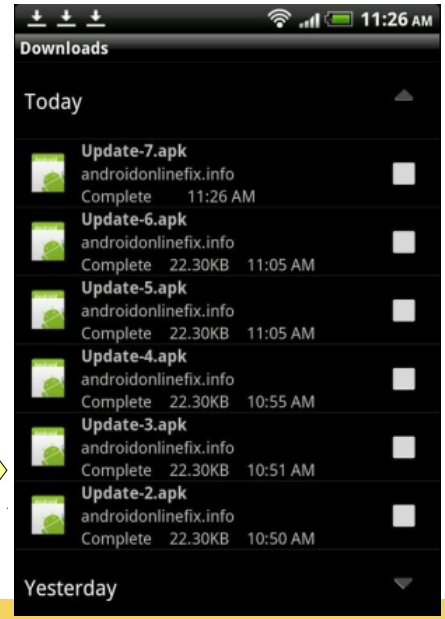
- Qui ha senso usare la registrazione dinamica
  - Registriamo il receiver quando avviamo un download
  - Lo de-registriamo al completamento
- In altri casi potrebbe essere meglio registrarlo staticamente (AndroidManifest.xml)

# Esempio: completamento download

- Il nostro receiver avrà:

```
public void onReceive(Context context, Intent i) {
 String act = i.getAction();
 if (DownloadManager.ACTION_DOWNLOAD_COMPLETE.equals(act)) {
  long id = i.getLongExtra(DownloadManager.EXTRA_DOWNLOAD_ID,0);
  // qui gestiamo il completamento del download id
 } else {
  // chissà, potremmo aver ricevuto altri intent
 }
}
```

Altri Intent utili
**ACTION_NOTIFICATION_CLICKED** – lanciato se l'utente seleziona la notifica del download
**ACTION_VIEW_DOWNLOADS** – da lanciare (via startActivity()) per attivare l'interfaccia del download manager stesso (“Downloads”)



# Esempio: accesso al file scaricato

- Il modo più semplice per accedere al file appena scaricato è di invocare alcuni metodi del DownloadManager stesso
- `ParcelFileDescriptor pfd = dm.openDownloadedFile(id);` restituisce un file descriptor (serializzabile) sul file scaricato, in sola lettura
- `Uri u = dm.getUriForDownloadedFile(id);` restituisce una URI sul file scaricato

# Accesso alle informazioni sui download

- Il DownloadManager mantiene un ContentProvider con le informazioni sui download
- Le applicazioni possono
  - Richiedere di aggiungere le informazioni sui propri download al provider
  - Fare query sul provider per avere informazioni sui file scaricati
- La classe DownloadManager.Query fornisce metodi di utilità per le query più frequenti

# Accesso alle informazioni sui download

- Per aggiungere un download alla gestione del DM

```
public long addCompletedDownload (
  String title,
  String description,
  boolean isMediaScannerScannable,
  String mimeType,
  String path,
  long length,
  boolean showNotification
)
```

- Fatto per default dal DM se non si chiedono opzioni particolari

# Esempio: accesso alle informazioni sui download

- Per recuperare informazioni dal ContentProvider:

```
Query q = new Query();
q.setFilterById(id);
Cursor c = dm.query(q);
if (c.moveToFirst()) {
  int j = c.getColumnIndex(DownloadManager.COLUMN_STATUS);
  int h = c.getColumnIndex(DownloadManager.COLUMN_LOCAL_URI);
  if (DownloadManager.STATUS_SUCCESSFUL == c.getInt(j)) {
    String uriString = c.getString(h);
    ImageView v =(ImageView)findViewById(R.id.v1);
    v.setImageURI(Uri.parse(uriString));
  }
}
```

Oppure: setFilterByStatus(...)

# Informazioni disponibili

| | |
|---|---|
| COLUMN_BYTES_DOWNLOADED_SO_FAR | Number of bytes download so far. |
| COLUMN_DESCRIPTION | The client-supplied description of this download. |
| COLUMN_ID | An identifier for a particular download, unique across the system. |
| COLUMN_LAST_MODIFIED_TIMESTAMP | Timestamp when the download was last modified, inSystem.currentTimeMillis() (wall clock time in UTC). |
| COLUMN_LOCAL_FILENAME | The pathname of the file where the download is stored. |
| COLUMN_LOCAL_URI | Uri where downloaded file will be stored. |
| COLUMN_MEDIAPROVIDER_URI | The URI to the corresponding entry in MediaProvider for this downloaded entry. |
| COLUMN_MEDIA_TYPE | Internet Media Type of the downloaded file. |
| COLUMN_REASON | Provides more detail on the status of the download. |
| COLUMN_STATUS | Current status of the download, as one of the STATUS_* constants. |
| COLUMN_TITLE | The client-supplied title for this download. |
| COLUMN_TOTAL_SIZE_BYTES | Total size of the download in bytes. |
| COLUMN_URI | URI to be downloaded. |

# AsyncPlayer

# AsyncPlayer

- Molte classi di sistema gestiscono il multithreading completamente al loro interno
- **AsyncPlayer** è una di queste
  - Riproduce file audio, ovviamente in background
  - Il thread non si vede, ma c'è

| | |
|---|---|
| Costruttore | AsyncPlayer ap = new AsyncPlayer(“thriller”) |
| Avvia la riproduzione | ap.play(ctx,uri,looping,outstream) |
| Ferma la riproduzione | ap.stop() |

STREAM_ALARM
STREAM_DTMF
STREAM_MUSIC
STREAM_NOTIFICATION
STREAM_RING
STREAM_SYSTEM
STREAM_VOICE_CALL
USE_DEFAULT_STREAM_TYPE

# AsyncPlayer

- L'uso da parte delle applicazioni è banale
  - … se sono richieste capacità più avanzate, c'è tutta l'architettura del MediaPlayer
    - La vedremo più avanti
- Ma saremmo in grado di implementare noi le stesse funzionalità?
  - Guardiamo il codice sorgente di AsyncPlayer!

**`AsyncPlayer.java`**