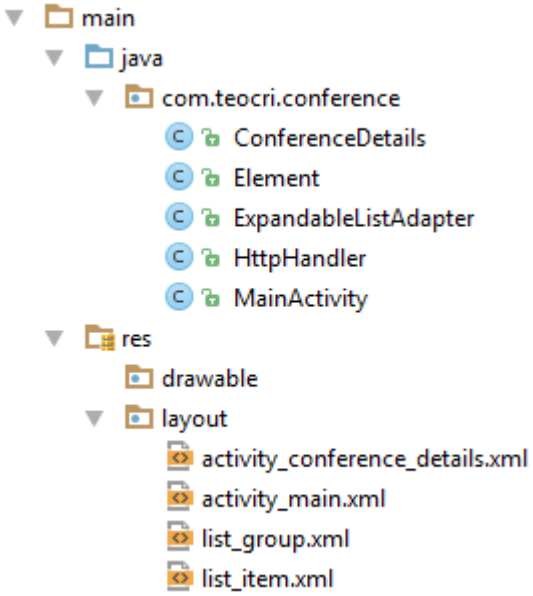
Relazione applicazioni mobili

Cacciamali Matteo: **20010709**
Nicolò Cristiano: **20011712**

Conference app:



MainActivity

La classe in questione contiene un ExpandableListView con cui mostriamo una lista delle singole date in cui abbiamo almeno una conferenza. Se espandiamo una di queste date avremo una vista su titolo e orario delle conferenze per quella data, ordinate per orario.
Il Main viene quindi usato per scaricare il file JSON da link firebase(usando un AsyncTask), leggere i dati da file, creare le liste di elementi mostrati nel ExpandableListView e startare la ConferenceDetails class.

ConferenceDetails

Questa classe mostrerà semplicemente in una nuova activity i dati relativi alla conferenza scelta e permetterà, tramite apposito bottone, l'inserimento dell'evento all'interno del calendario.

ExpandableListAdapter

Serve a gestire l'utilizzo del ExpandableListView, per mostrare a video le diverse conferenze.

Element

Abbiamo creato questa classe per poter avere tutte le informazioni in una sola struttura da passare dal MainActivity alla ConferenceDetails in modo semplice tramite: intent.putExtra() (permesso dal fatto che Element implementa Serializable).

HttpHandler

Questa classe ha il compito di creare la connessione tramite url e effettuare la GET del file JSON da cui leggiamo i dati nel MainActivity.

Parti ambigue di codice:

```
for (int i = 0; i < listGroup.size(); i++){
    List<Element> tmp = new ArrayList<>();
    for (int j = 0; j < elements.size(); j ++){
        if (elements.get(j).getDate().trim().equals(listGroup.get(i).trim())) {
            tmp.add(elements.get(j));
            Collections.sort(tmp, (Comparator) (e1, e2) → {
                    return Integer
                            .valueOf(getNumericTimeValue(e1.getTime()))
                            .compareTo(Integer.valueOf(getNumericTimeValue(e2.getTime())));
            });
        }
    }
    listChild.put(listGroup.get(i), tmp);
}
```

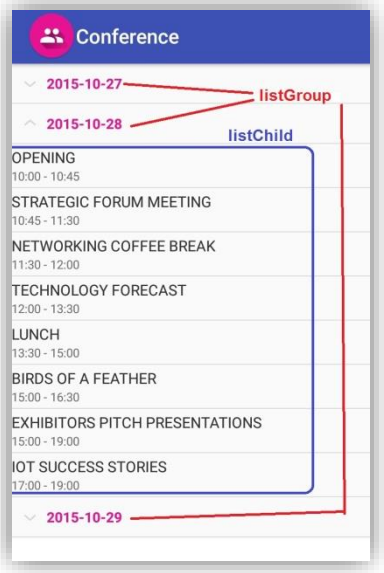
Questo ciclo situato all'interno del AsyncTask, appena dopo la lettura dei dati, serve a fornire i dati in modo strutturato per l'ExpandableListView. Abbiamo ragionato nel seguente modo:

Primo FOR:
Per ogni singola data trovata durante la lettura (listGroup.size) creiamo una List<Element> temporanea (dato che la listChild richiede una List<Element> come secondo parametro).

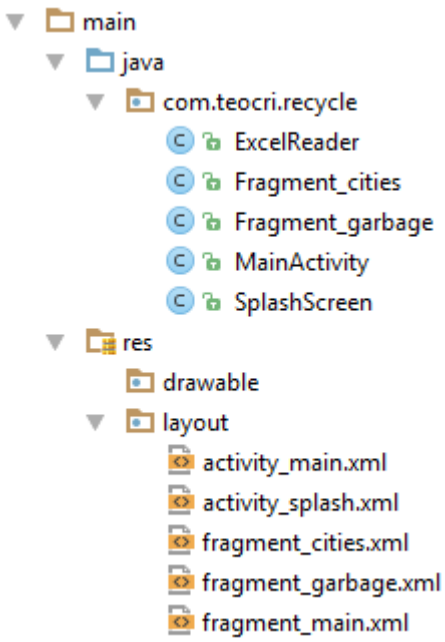
Secondo FOR:
Una volta creata la nostra lista temporanea guardiamo per ogni conferenza letta da file(indice **j**), se appartiene o meno alla singola data che stiamo analizzando (indice **i**).

IF dentro il secondo FOR:
Se la data della conferenza **j** che stiamo controllando equivale alla data **i** => questa conferenza andrà come figlio di questa data nel ExpandableListView, altrimenti si prosegue con i cicli.

Collections.sort:
Questa parte ha solo il compito di ordinare temporalmente le conferenze nella lista temporanea.



Recycle app:



SplashScreen

Serve a far ritardare di qualche secondo la visualizzazione del main, per far sapere all'utente che serve qualche istante a caricare i dati. Per farlo partire come pagina iniziale anzichè il Main abbiamo inserito questa parte di codice nel manifest

```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

MainActivity

Il MainActivity conterrà semplicemente i metodi dei bottoni e il SectionsPagerAdapter per la gestione del viewPager grazie al quale si può fare swipe tra le 2 pagine.

Fragment_cities & Fragment_garbage

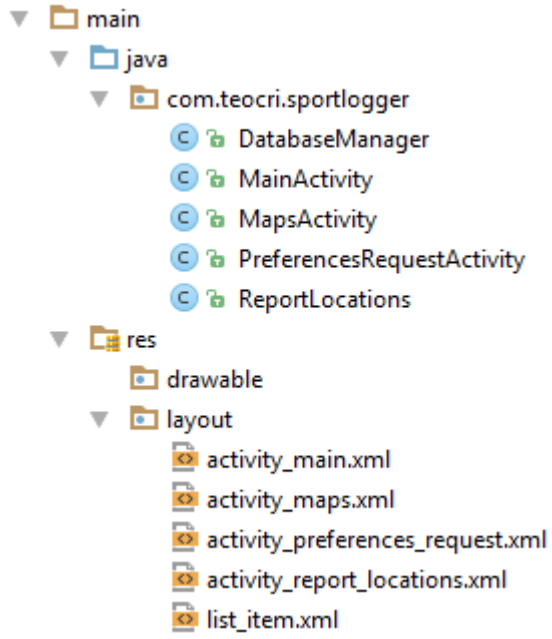
Queste due classi contengono solamente i costruttori che servono nel SectionsPagerAdapter.

```
@Override
public Fragment getItem(int position) {
    if (position == 0)
        return Fragment_cities.newInstance();
    return Fragment_garbage.newInstance();
}
```

ExcelReader

Questa classe effettua la lettura dal file elenco.xlsx presente nella cartella raw e contiene i due metodi: readSchedulePerCity e readRecycleBin che vengono usati nei bottoni per ricevere i risultati della lettura da file a seconda dell'input.

Sport logger:



MainActivity

Questa volta anzi che la Splash screen usiamo il Main come pagina di caricamento iniziale, dato che facciamo nel main la *db* = **new** DatabaseManager(**this**). Dopo 1 secondo avviamo la MapsActivity.

MapsActivity

Questa è la classe con cui l'utente si interfaccerà con mappa, streetView e tutte le funzionalità offerte dal menu. Questa classe effettua quindi tutte le operazioni di connessione, disconnessione, riconnessione e controllo dei permessi per la localizzazione del dispositivo. Setta in oltre alcuni parametri iniziali come tempo e displacement che potranno essere modificati dalla apposita activity di impostazioni/preferences. Effettua anche tutte le intent per accedere alle altre 2 activities PreferencesRequestActivity e ReportLocation. La variabile reconnecting, il cui utilizzo può sembrare strano, serve a fare si che se si sta effettuando una riconnessione allora non si dovrà mettere subito il primo marker iniziale(come viene fatto solitamente quando si effettua la connessione), così da evitare che l'utente possa mettere troppi marker in un solo punto continuando a cambiare impostazioni/preferences (motivazione della riconnessione spiegata nella PreferencesRequestActivity).

PreferencesRequestActivity

Questa classe permette di modificare il tempo e il displacement con cui verranno aggiunti i markers sulla mappa. Va detto che se delle impostazioni vengono modficate, al ritorno nella MapsActivity quest'ultima provvederà a disconnettersi e riconnettersi automaticamente al servizio di localizzazione per utilizzare le nuove impostazioni appena modificate.

ReportLocations

Questa activity, tramite una query al db, mostra le singole date in cui sono stati piazzati dei markers sulla mappa e il numero di markers per quel dato giorno. I giorni sono ordinati per data crescente e cliccando sopra una di queste date si muoverà la mappa di conseguenza al primo marker di quel giorno.

DatabseManager

Questa classe serve semplicemente a gestire il db per creazione, inserimento, cancellazione e query.

Parti ambigue di codice:

```
@Override
public void onStreetViewPanoramaCameraChange(StreetViewPanoramaCamera streetViewPanoramaCamera) {
    if (upDownSide == false)
        mMap.moveCamera(
                CameraUpdateFactory.newCameraPosition
                        (new CameraPosition(mMap.getCameraPosition().target,
                                            mMap.getCameraPosition().zoom,
                                            mMap.getCameraPosition().tilt,
                                            mStreet.getPanoramaCamera().bearing)));
}

@Override
public void onCameraMove() {
    if (upDownSide == true)
        mStreet.animateTo(
                new StreetViewPanoramaCamera(
                        mStreet.getPanoramaCamera().zoom,
                        mStreet.getPanoramaCamera().tilt,
                        mMap.getCameraPosition().bearing), -1);
}
```

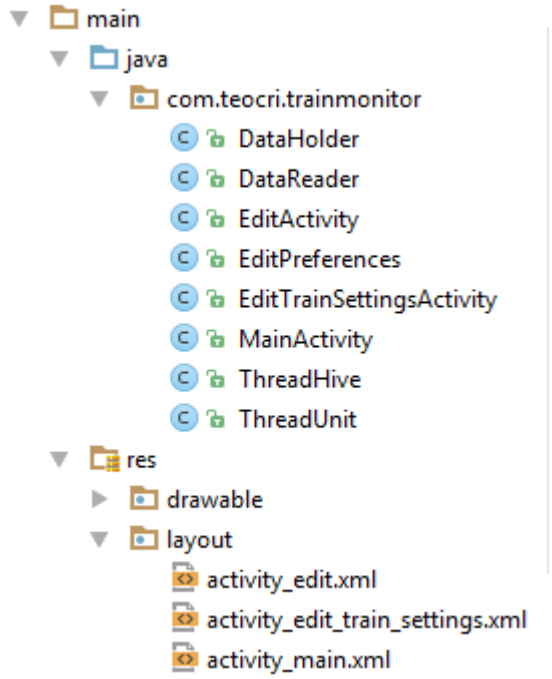
Questa parte di codice ci serve a risolvere un problema che abbiamo incontrato cercando di far muovere la mappa quando l'utente muove la street view o viceversa.

Il problema che avevamo è che muovendo una delle due l'altra si muoveva a scatti, probabilmente a causa del fatto che funzionando assieme, ognuna cercava di muovere di conseguenza l'altra.

SOLUZIONE:

quello che abbiamo fatto è stato aggiungere un Frame Layout(invisibile all'utente) nella MapsActivity come ultimo elemento in modo che fosse sopra a tutto il resto, così da poter prendere l'evento di touch da parte dell'utente (dato che sui fragments non riuscivamo al meglio). Una volta intercettato l'evento di touch controlliamo se è avvenuto ad una "altezza" superiore a quella della street view (quindi sulla mappa) o viceversa così da far funzionare i due metodi in figura uno alla volta e mai in contemporanea.

Train monitor:



MainActivity

Il MainActivity è la classe con cui l'utente potrà monitorare la situazione dei treni a video. Da questa activity vengono gestite le chiamate alle altre activity e il menu.

DataHolder

Questa è una classe statica che abbiamo usato per tenere ordinate tutte le informazioni e i riferimenti ai dati dei treni così che da ogni altra activity sia possibile accedere a tali dati in modo semplice e chiaro.

DataReader

Questa classe contiene i metodi per connessione e lettura dei dati dalla pagina HTML da cui vengono prese le informazioni sui treni.

EditActivity

Questa è la classe che verrà chiamata dal MainActivity quando l'utente preme su edit/modifica in uno slot vuoto. Quello che fa quindi è chiedere un numero di treno in input e il tempo di aggiornamento; effettua i controlli di input vuoto o di input già monitorato, in caso contrario effettua l'aggiunta di un nuovo treno con l'utilizzo del DataHolder.

EditTrainSettingsActivity

Questa activity viene (contrariamente alla precedente) chiamata quando l'utente preme su edit/modifica quando lo slot non è vuoto. Permette semplicemente di modificare l'intervallo di aggiornamento del treno o di eliminarlo.

ThreadHive

Questa è la classe per creare, eliminare, eseguire, restartare e fermare i ThreadUnit.

ThreadUnit

Questa classe estende la classe AsyncTask e serve per effettuare la connessione tramite il DataReader e gestire l'aggiornamento dei dati su monitor ogni tot polling time scelto dall'utente.

Note sul train monitor

Abbiamo notato che su alcuni dispositivi in cui abbiamo testato la nostra app solo 5 threads lavorano in contemporanea, mentre il sesto viene inserito in coda. Abbiamo fatto delle ricerche e abbiamo visto che pare essere un problema comune relativo agli AsyncTask. Ne avevamo parlato anche per mail e ci aveva consigliato di provare una possibile soluzione che abbiamo provato, ma purtroppo non ha risolto il problema.