Modulo 7

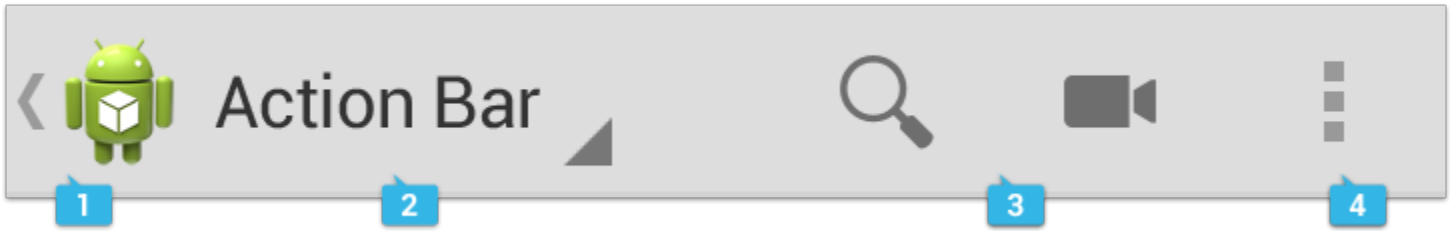
Programmazione avanzata in Android

7.1

Action Bar e navigazione

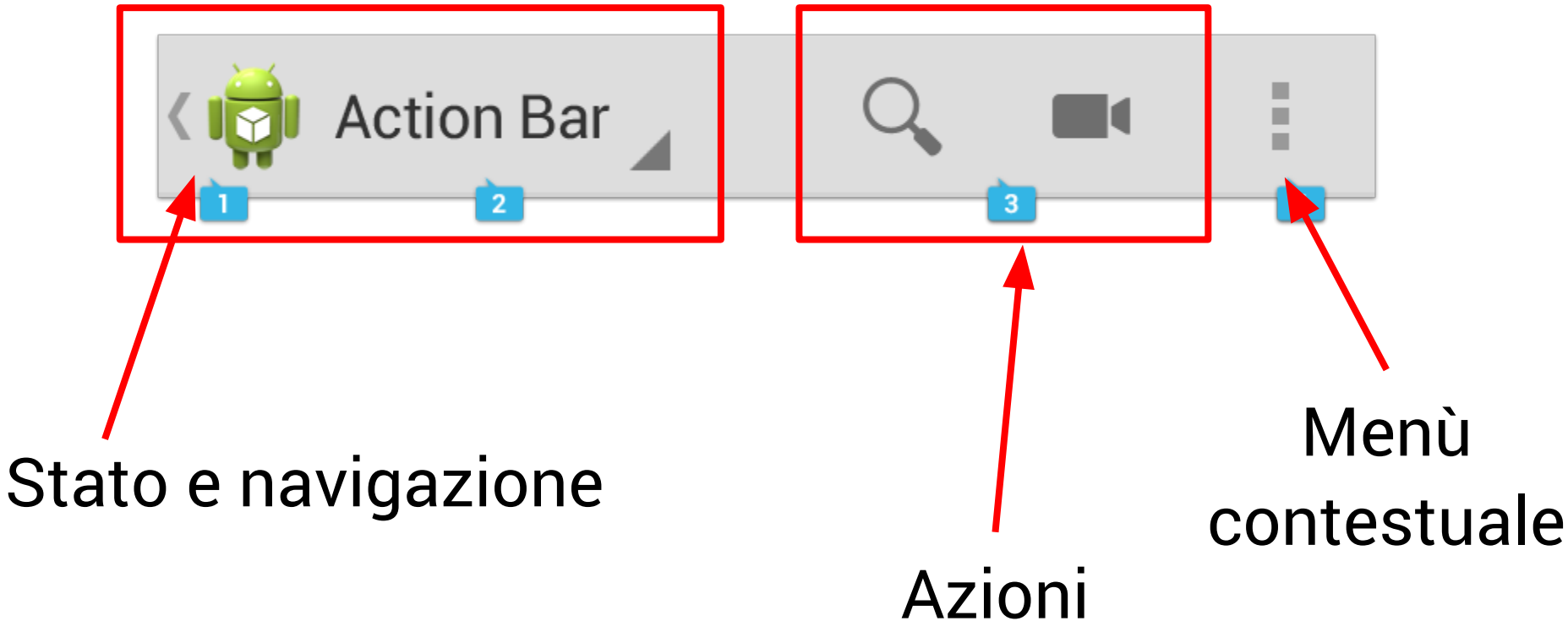
7.1 Action Bar

Novità di Android 3.0 *Honeycomb* (API lvl 11).



7.1 Action Bar

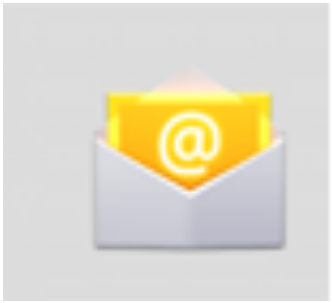
Novità di Android 3.0 *Honeycomb* (API lvl 11).



7.1 Action Bar per navigazione

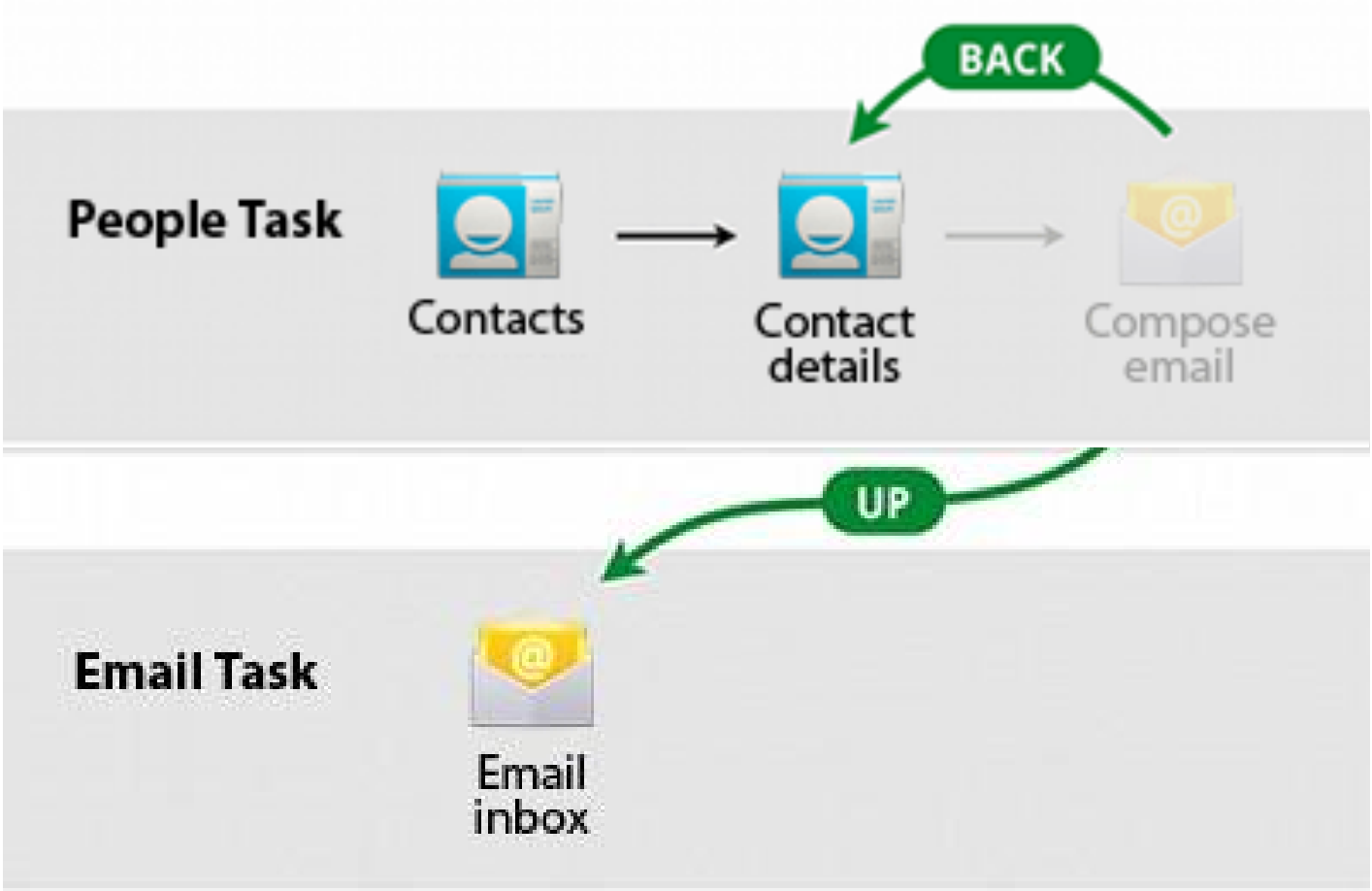
Nuovo modello di navigazione da *Honeycomb*:

Action Bar rappresenta graficamente anche lo stack di navigazione.



7.1 Nuovo modello di navigazione

7.1 Nuovo modello di navigazione



7.1 Task stack "sintetico"

Si utilizza un `TaskStackBuilder`.

(Nella support library.)

Genera uno stack "sintetico" su Android ≥ 3.0.

Non genera nessuno stack e funziona in maniera *legacy* su Android < 3.0.

7.1 Demo

Action Bar

Navigazione semplice

Navigazione "Up"

7.1 Intent "up"

Per generare l'Intent di navigazione:

```
NavUtils.navigateUpFromSameTask()
```

7.1 Intent "up"

Per generare l'Intent di navigazione:

~~`NavUtils.navigateUpFromSameTask()`~~

...e se *non* siamo nello stesso Task?!

7.1 Intent "up"

Per generare l'Intent di navigazione:

~~`NavUtils.navigateUpFromSameTask()`~~

Si utilizza nuovamente un `TaskStackBuilder`:

Si genera un *task sintetico* ***nuovo***,
rigenerando lo stato dell'applicazione.

Faticoso? *Sì*.

7.1 Demo

Navigazione "Up" complessa
con Intent filter

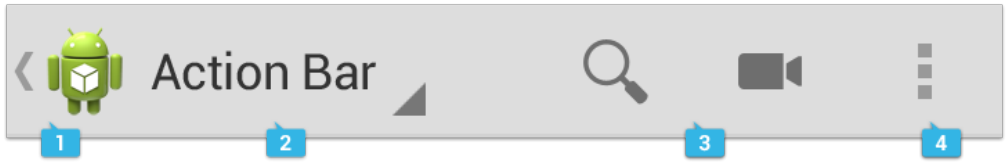
7.1 Pulsanti nella Action Bar

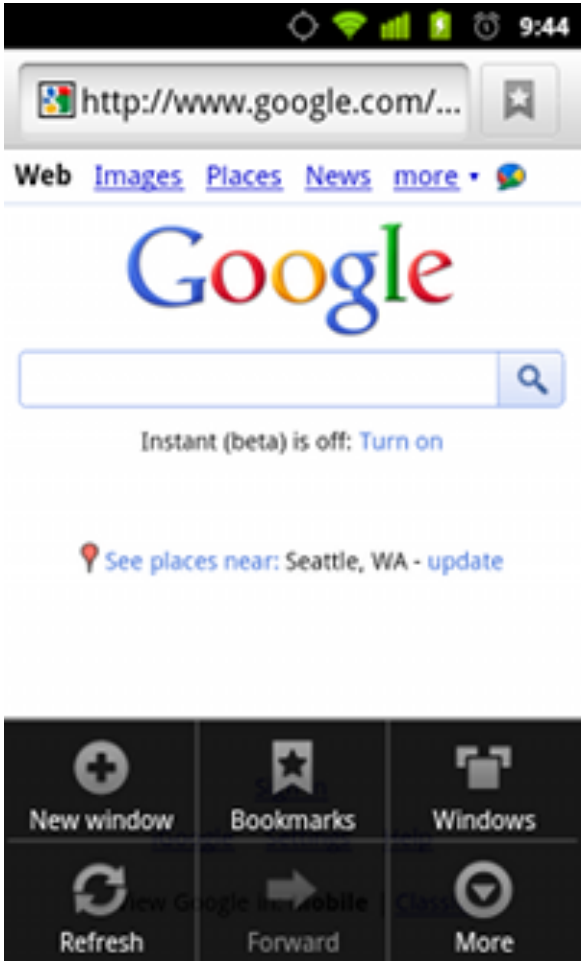
Conversione del "vecchio" menu contestuale di Android pre-*Honeycomb*.

Viene "gonfiato" come il layout principale:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
   getMenuInflater()
      .inflate(R.menu.main, menu);
   return true;
}
```

7.1 Retrocompatibilità dei menu





7.1 Definizione dei menu

Risorsa XML in cartella `/res/menu`:

```
<menu>
   <item
      android:id="@+id/action_ciao"
      android:showAsAction="always|withText"
      android:title="Ciao mondo!"
      android:titleCondensed="Ciao"
      android:icon="@drawable/ciao"
      />
```

7.1 Definizione dei menu

Risorsa XML in cartella `/res/menu`:

```
<menu>
    <item
        android:id="@+id/action_ciao"
        android:showAsAction="always|withText"
        android:title="Ciao mondo!"
        android:
        android:
        />
```

- always
- never
- ifRoom
- withText

7.1 Eventi dei menu

Metodo virtuale della classe `Activity`:

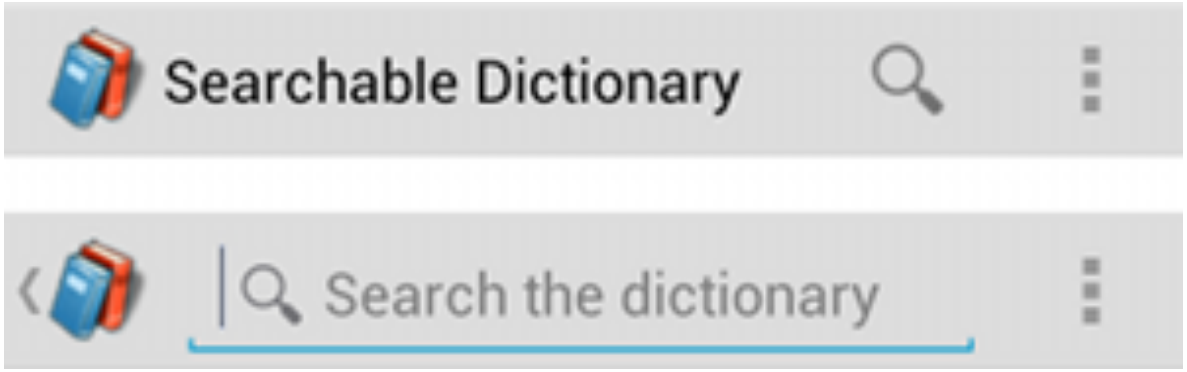
```
@Override
public boolean onOptionsItemSelected(MenuItem
item) {
   switch(item.getItemId()){
      case R.id.action_ciao:
         //...
         break;
```

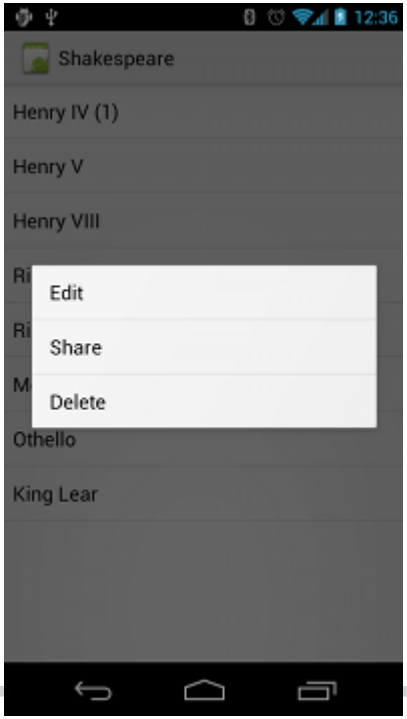
7.1 Demo

Action Bar, Notifiche e TaskStackBuilder.

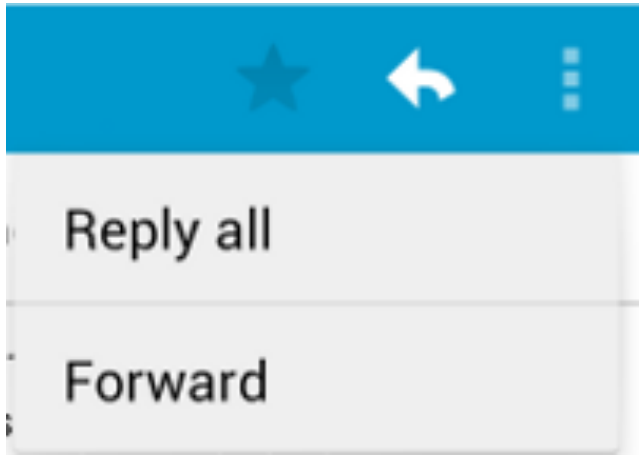
7.1 Altre cose dei menu...

actionViewClass





Context menu



Popup menu

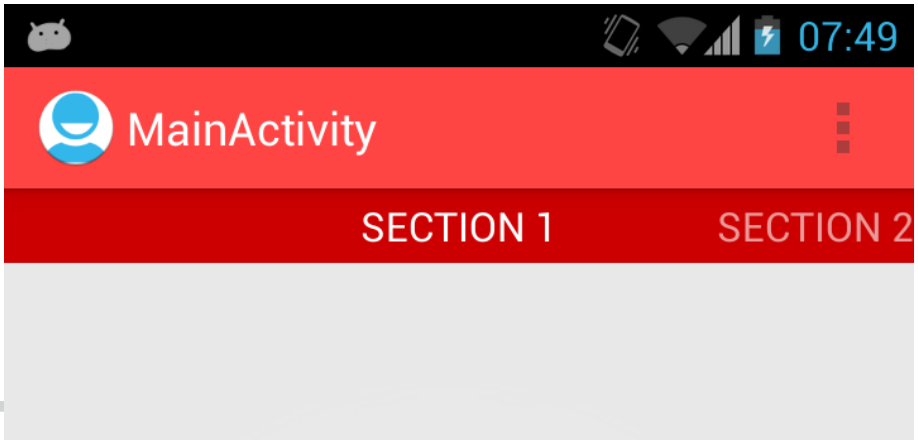
7.1 Navigazione tab based

Tab nell'Action Bar: sfrutta Fragment nativi.

(API level 11.)

Soluzione compatibile:

- PagerTitleStrip
- PagerTabStrip



7.1 Demo

ViewPager, PagerAdapter e PagerTitleStrip.

7.2

Notifiche avanzate

7.2 Demo

Notifica "speciale".

7.2 Demo

Stili estesi delle notifiche:

`BigText`

`BigPicture`

`Inbox`

7.2 Demo

Notifiche di progresso.

7.3

Threading in Android

7.3 Threading

Ogni applicazione Android vive in **1 processo**.

(Activity, Servizi, etc...)

Ogni processo Android nasce con **1 thread**.

Noto come "*UI Thread*".

(Esistono thread aggiuntivi per la GC, ma non sono visibili al programmatore.)

7.3 Multithreading

Android ha *molte* primitive per il multithreading.

- Thread (Java)
- Timer (Java)
- AsyncTask (Android)
- ThreadPool (Java)

Sono tutte a livello di *classi*. Scarso supporto nel linguaggio stesso. (`synchronized`)

7.3 Attenzione ai servizi!

I servizi **non** vivono in un loro thread!

Condividono lo *UI thread* di eventuali `Activity` nella stessa applicazione. Codice bloccante è bloccante per *tutte* le `Activity` e *tutti* i `Service`.

Un `Service` ha un *ciclo di vita* separato.

7.3 Demo

Thread background,
con accesso UI

7.3 Multithreading UI

Il toolkit UI di Android ***non*** è *thread-safe*!

(Nessun toolkit lo è, di solito...)

Accesso all'UI (classi `View`) ***deve*** avvenire soltanto da parte del *UI Thread*.

7.3 Multithreading UI

Il toolkit di Android mette a disposizione i seguenti concetti primitivi:

- `Looper`
- `Handler`
  - Gestiscono blocchi di codice encapsulati dall'interfaccia `Runnable`

7.3 Demo

Thread

Timer

AsyncTask

e accesso UI corretto

7.5

Richieste HTTP e download

7.5 Due implementazioni

Android contiene ben 2 implementazioni complete del protocollo HTTP:

- Client Apache:

  Interfaccia complessa, flessibile, estremamente versatile e completa.

  `HttpClient` e `AndroidHttpClient`.

- Client Android:

  Semplice, facile da utilizzare, più limitato... e buggato.

  `HttpURLConnection`.

7.5 Due implementazioni

Android contiene ben 2 implementazioni complete del protocollo HTTP:

- Client Apache:

  Interfaccia complessa, flessibile, estremamente versa

  `Http`

  OK, nell'80% dei casi...

- Client Android:

  Semplice, facile da utilizzare, più limitato... e buggato.

  `HttpURLConnection`.

7.3 Comodità

Compressione GZip automatica.

Caching locale *non* funziona! (Manuale < 4.2)

```
.setUseCaches(true);
```

Facile configurazione della richiesta:

```
.addRequestProperty(field, value);
.setRequestMethod("POST");
.getOutputStream();
```

7.5 Esecuzione di una richiesta

```
.connect();

.getResponseCode() == HTTP_OK

InputStream s = .getInputStream();
s.Read(...);

.disconnect();
```

7.5 HTTP e multithreading

Da Android 4.1 è **vietato** effettuare richieste HTTP dal thread principale (*UI thread*).

Utilizzare *sempre* un thread separato, ad esempio con un `AsyncTask`.

7.5 DownloadManager

Servizio di sistema di Android, che permette di avviare dei download via HTTP.

- Funziona in background
- Supporta il *resume*
- Controllabile dall'utente
- Affidabile
- Comunica via broadcast intent

7.5 Demo

DownloadManager

7.6

Parcellizzazione

7.6 Parcellizazione

Tecnica Android per il trasporto IPC ad alte prestazioni ed a basso livello: l'implementazione sottostante potrebbe cambiare nel tempo quindi non può essere utilizzata come tecnica di serializzazione per la persistenza.

7.6 Parcellizzazione

Permette di gestire 6 classi di dati:

1. **Primitivi**
2. **Array primitivi**
3. **Parcellizzabili**
4. Bundles
5. Oggetti attivi
6. Contenitori non tipizzati

7.6 Parcellizzazione - primitivi

Scrittura	Lettura
writeBytes(byte)	*readBytes()*
writeDouble(double)	*readDouble()*
writeFloat(float)	*readFloat()*
writeInt(int)	*readInt()*
writeLong(long)	*readLong()*
writeString(String)	*readString()*

7.6 Parcellizazione - array primitivi

Scrittura	Lettura
writeBooleanArray(boolean[])	*readBooleanArray(boolean[])*
writeByteArray(byte[])	*readByteArray(byte[])*
writeCharArray(char[])	*readCharArray(char[])*
writeDoubleArray(double[])	*readDoubleArray(double[])*
writeFloatArray(float[])	*readFloatArray(float[])*
writeIntArray(int[])	*readIntArray(int[])*
writeLongArray(long[])	*readLongArray(long[])*
writeStringArray(String[])	*readStringArray(String[])*

7.6 Parcellizzazione - parcellizzabili

Scrittura	Lettura
writeParcelable(Parcelable, int)	*readParcelable(ClassLoader)*
writeParcelableArray(T[], int)	*readParcelableArray(ClassLoader)*

È una maniera estremamente efficiente per rendere auto-serializzanti/deserializzanti degli oggetti tramite la semplice implementazione di un'interfaccia Java.

7.7

Applicazioni che si adattano a tablet e smartphone

7.7 Applicazioni smartphone-tablet

L'idea alla base dell'applicazione unificata tra tablet e smartphone è l'utilizzo dei fragment.

7.7 Applicazioni smartphone-tablet

In particolare verrà utilizzato l'attach statico per la versione tablet del layout, mentre nella versione smartphone l'attach dinamico è il preferito.

7.7 Applicazioni smartphone-tablet

All'activity è lasciato il ruolo di coordinatrice per la logica inserita direttamente dentro i fragments.

7.8

Persistenza

7.8 Persistenza

Android mette a disposizione diverse metodologie per la gestione della persistenza:

- SharedPreferences
- SQLite
- File System

7.8.1 SharedPreferences

Le abbiamo già incontrate nei moduli precedenti ma le affrontiamo di nuovo per capire meglio le altre opzioni di storage.

Rappresentano delle preferenze che devono essere accessibili nell'applicazione per personalizzare l'esperienza d'uso dell'utente.

7.8.1 SharedPreferences

Internamente vengono salvate nella cartella privata dell'applicazione in un file XML in cui ogni nodo rappresenta una coppia chiave/valore.

Possono essere resi persistenti tutti i tipi di dati primitivi quindi *int*, *long*, *float*, *double*, *boolean*, *string*.

7.8.1 SharedPreferences

Per ottenere il riferimento alla preferenza è necessario chiamare il metodo getSharedPreferences su un oggetto Context.

```
public class Calc extends Activity {
        public static final String PREFS_NAME = "MyPrefsFile";

        public void test(){
                SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
        }
}
```

7.8.1 SharedPreferences

È possibile interrogare l'oggetto recuperato tramite i metodi ***getXXX(chiave, default)*** sostitendo ad XXX il tipo di dato da recuperare.

```
public void test(){
        SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
        boolean saveOnExit = settings.getBoolean("saveOnExit", false);
}
```

PS: se il tipo di dato salvato nella preferenza non corrisponde con il metodo utilizzato per il recupero, viene lanciata un'eccezione.

7.8.1 SharedPreferences

Per modificare l'oggetto recuperato è necessario ottenere un suo ***Editor*** tramite il quale sarà possibile inserire/modificare le coppie chiave/valore ed effettuare il commit delle modifiche apportate.

```
public void saveOnExit(boolean save){
        SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
        SharedPreferences.Editor editor = settings.edit();
        editor.putBoolean("saveOnExit", save);
        editor.commit();
}
```

7.8.2 SQLite

Se abbiamo una notevole quantità di informazioni che vogliamo rendere persistenti ed interrogabili in maniera semplice ed efficiente, Android ci consente di utilizzare al meglio un database relazionale tramite SQLite.

7.8.2 SQLite

SQLite è un database engine leggero (<500kb) e opensource che opera direttamente su un file: ciò vuol dire che i database creati possono essere copiati e spostati tra dispositivi diversi semplicemente spostando il file stesso.

Supporta tabelle, viste, indici, transazioni, trigger, etc.

7.8.2 SQLite

Quando l'app crea un database, viene creato il file corrispondente in posizione:

DATA/data/<app_package>/databases/NAME

DATA è il path ritornato dal metodo *Environment.getDataDirectory()*
NAME è il nome del database creato tramite i metodi che vedremo fra poco

7.8.2 SQLite

Per creare, aprire ed effettuare l'upgrade di un database è consigliato estendere un SQLiteOpenHelper (classe astratta) implementando i metodi onCreate ed onUpgrade.

7.8.2 SQLiteOpenHelper

```
public class DatabaseHelper extends SQLiteOpenHelper {

        private static final String DATABASE_NAME = "mydatabase.db";
        private static final int DATABASE_VERSION = 1;

        private static final String DATABASE_CREATE = "CREATE TABLE contact (_id INTEGER
PRIMARY KEY AUTOINCREMENT, name TEXT NOT NULL, email TEXT NOT NULL, gender TEXT NOT
NULL);";

        public DatabaseHelper(Context context) {
                super(context, DATABASE_NAME, null, DATABASE_VERSION);
        }

        // ...
}
```

7.8.2 SQLiteOpenHelper

```
public class DatabaseHelper extends SQLiteOpenHelper {
        // ..

        // Questo metodo viene chiamato durante la creazione del database
        @Override
        public void onCreate(SQLiteDatabase database) {
                database.execSQL(DATABASE_CREATE);
        }

        // Questo metodo viene chiamato durante l'upgrade del database,
        // tipicamente quando viene incrementato il numero di versione
        @Override
        public void onUpgrade(SQLiteDatabase database, int oldVersion, int newVersion) {
                database.execSQL("DROP TABLE IF EXISTS contact");
                onCreate(database);
      }

}
```

7.8.2 SQLite

Tramite l'SQLiteOpenHelper è possibile ottenere un'istanza in sola lettura o in scrittura del database su cui operare:

```
DatabaseHelper helper = new DatabaseHelper(this);
// ottenere un'istanza in sola lettura
SQLiteDatabase database = helper.getReadableDatabase();
// ottenere un'istanza in scrittura
SQLiteDatabase database = helper.getWritableDatabase();
```

7.8.2 SQLite - CRUD (Create)

Tramite l'istanza SQLiteDatabase possiamo facilmente creare una riga in una tabella utilizzando un oggetto ContentValues ed il metodo ***insert*** della classe SQLiteDatabase :

```
ContentValues values= new ContentValues();
values.put("id", 1);
values.put("name", "Mario");
// ..
SQLiteDatabase database = helper.getWritableDatabase();
database.insert("contact", null, values);
```

7.8.2 SQLite - CRUD (Retrieve)

Per ricavare invece una riga dal database ci è messo a disposizione l'oggetto Cursor ed il metodo ***query***, che tra gli altri richiede quali parametri la tabella su cui effettuare la ricerca e le condizioni che la query deve rispettare:

```
// firma del metodo query della classe SQLiteDatabase
Cursor query(String table, String[] columns, String selection, String[]
selectionArgs, String groupBy, String having, String orderBy, String limit);
```

7.8.2 SQLite - CRUD (Retrieve)

```
Contact contact = null;
Cursor cursor = database.query("contact", null, "_id = ?", new String[] {"1"},
                               null, null, null);
// il metodo query ritorna un oggetto Cursor che possiamo scorrere, una volta
che l'abbiamo posizionato in prima posizione.
if(cursor.moveToFirst()){
      contact = new Contact();
      // otteniamo i dati delle colonne della tabella a partire dai loro indici
      contact.setId(cursor.getLong(0));
      contact.setName(cursor.getString(1));
      // ..
}
```

7.8.2 SQLite - CRUD (Update)

L'update funziona in maniera simile al create, con la differenza che è necessario specificare quali righe aggiornare utilizzando il metodo ***update*** anzichè il metodo ***create***.

```
// firma del metodo update della classe SQLiteDatabase
int update (String table, ContentValues values, String whereClause, String[]
whereArgs);
```

7.8.2 SQLite - CRUD (Update)

```
ContentValues values= new ContentValues();
values.put("id", 1);
values.put("name", "Nuovo nome");
// specifichiamo di aggiornare le righe in cui l'id è uguale a 1
database.update("contact", values, "_id = ?", new String[] {"1"});
```

7.8.2 SQLite - CRUD (Delete)

Cancellare è un'operazione ancora più semplice delle altre, perché è sufficiente specificare al metodo ***delete*** le condizioni che devono essere soddisfatte dalle righe da eliminare.

```
// firma del metodo delete della classe SQLiteDatabase
int delete (String table, String whereClause, String[] whereArgs);
```

7.8.2 SQLite - CRUD (Delete)

```
String conditions = "name = ?";
String[] conditionsValues = new String[] {"Mario"};
// specifichiamo di aggiornare le righe in cui l'id è uguale a 1
database.delete("contact", conditions, conditionsValues);
```

7.8.2 SQLite - Transazioni

Tramite l'oggetto SQLiteDatabase è possibile gestire esplicitamente le transazioni:

```
try{
	// dapprima avviamo la transazione.
	database.beginTransaction();
	// facciamo qualcosa di complesso;
	// ...
	// quindi effettuiamo il commit della transazione
	database.setTransactionSuccessful();
}finally{
	// quando chiudiamo la transazione, se non è stato effettuato il commit,
	// il rollback viene effettuato di default.
	database.endTransaction();
}
```

7.8.3 File System

Quando vogliamo salvare una quantità elevata di dati in cui nè il modello chiave/valore nè quello relazionale si rivelano utili, possiamo utilizzare direttamente il salvataggio su file.

Android espone un file system completo alle applicazioni con l'unica limitazione data dai permessi ristretti sulle cartelle/file non pubblici (tipicamente i dati di altre app).

7.8.3 File System - internal storage

Una possibilità che abbiamo è quella di salvare i file nello spazio privato dell'applicazione, eventualmente gestendo i permessi per l'accesso degli altri utenti (app e utente stesso).

In questa maniera i file saranno automaticamente eliminati quando l'utente disinstalla la nostra app.

7.8.3 File System - internal storage

Per creare un file l'oggetto Context espone il metodo ***openFileOutput:***

```
FileOutputStream openFileOutput (String name, int mode);
```

Le modalità di apertura disponibili sono:

- **MODE_PRIVATE**
- MODE_APPEND
- MODE_WORLD_READABLE
- MODE_WORLD_WRITEABLE

7.8.3 File System - internal storage

Sull'oggetto FileOutputStream ritornato possiamo agire come nel classico Java, utilizzando quindi il metodo ***write*** per la scrittura e il metodo ***close*** finale per chiudere il file:

```
FileOutputStream fos = openFileOutput("nome.txt", Context.MODE_PRIVATE);
fos.write("prova".getBytes());
fos.close();
```

7.8.3 File System - internal storage

Specularmente per aprire un file e leggerlo abbiamo il metodo ***openFileInput***:

FileInputStream **openFileInput** (String name);

NB: se il file specificato non esiste viene lanciata un'eccezione di tipo FileNotFoundException.

In questo caso viene ritornato un oggetto FileInputStream che espone il metodo ***close*** che questa volta seguirà il metodo ***read*** utilizzato per leggere da file.

7.8.3 File System - external storage

Tutti i dispositivi Android supportano uno storage esterno condiviso che può essere rimovibile (e.g. una scheda SD) o non rimovibile (storage interno).

I file creati nello spazio esterno sono modificabili da tutti, compreso l'utente quando collega il device come unità di storage USB.

7.8.3 File System - external storage

Come se non bastasse un utente può smontare il volume esterno runtime, quindi le probabilità che lettura/scrittura non vadano a buon fine non sono poi così remote.

Attenzione!

7.8.3 File System - external storage

Prima di procedere all'utilizzo dello spazio esterno dobbiamo verificare che lo stato in cui si trova sia adeguato.

```
String state = Environment.getExternalStorageState();
if (Environment.MEDIA_MOUNTED.equals(state)) {
        // Possiamo scrivere sull'unità esterna
} else if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        // Possiamo leggere dall'unità esterna
}else { // Non possiamo nè leggere nè scrivere sull'unità esterna }
```

7.8.3 File System - external storage

Le operazioni di lettura e scrittura su file sull'unità esterna vengono effettuate nella classica maniera Java tramite l'utilizzo degli StreamWriter/StreamReader:

```
FileInputStream fin = new FileInputStream(new File(filePath));
BufferedReader reader = new BufferedReader(new InputStreamReader(fin));
StringBuilder sb = new StringBuilder();
String line = null;
while ((line = reader.readLine()) != null)
        System.out.println(line);
```

7.9

Alarm Manager

7.9 AlarmManager

È un servizio di sistema che permette di schedulare l'invio di Intent a certe scadenze e intervalli.

Viene utilizzato per limitare il tempo di vita dell'applicazione, in modo da consumare meno risorse energetiche e computazionali.

7.9 AlarmManager

Per esempio potrebbe essere utilizzato per controllare periodicamente la disponibilità di aggiornamenti dalla rete:

```
public class MyActivity extends Activity{
        ...
        // creiamo il PendingIntent da passare all'AlarmManager
        Intent intent = new Intent(this, MyService.class);
        PendingIntent pIntent = PendingIntent.getService(this, 0, intent, 0);
        ...
}
```

7.9 AlarmManager

Per esempio potrebbe essere utilizzato per controllare periodicamente la disponibilità di aggiornamenti dalla rete:

```
public class MyActivity extends Activity{
...
// quindi scheduliamo l'invio del PendingIntent
AlarmManager a = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
long now = System.currentTimeMillis();

a.setRepeating(AlarmManager.RTC_WAKEUP, now , 60 * 60 * 1000, pIntent);

}
```

7.9 AlarmManager

Per evitare di avere tanti eventi vicini tra di loro nel tempo ma non completamente coincidenti, Android permette di specificare intervalli ***inesatti***:

- INTERVAL_FIFTEEN_MINUTES
- INTERVAL_HALF_HOUR
- INTERVAL_HOUR
- INTERVAL_HALF_DAY
- INTERVAL_DAY

7.9 AlarmManager

Per evitare di avere tanti eventi vicini tra di loro nel tempo ma non completamente coincidenti, Android permette di specificare intervalli ***inesatti***:

```
public class MyActivity extends Activity{

AlarmManager a = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
long now = System.currentTimeMillis();

a.setInexactRepeating(AlarmManager.RTC_WAKEUP, now,

                    AlarmManager.INTERVAL_HOUR, pIntent);

}
```

7.10

Location API

7.10 Location API

Android mette a disposizione classi e metodi per ottenere la ***posizione geografica attuale***, essere notificati ai ***cambi di posizione***, trovare latitudine e longitudine dato un indirizzo (***forward geocoding***), ricavare un indirizzo dati latitudine e longitudine (***reverse geocoding***).

7.10 Location API

Posizione geografica attuale:

```
// Acquisiamo il servizio di sistema per localizzarsi
LocationManager lm = (LocationManager) getSystemService
                                (Context.LOCATION_SERVICE);
// Definiamo un criterio per avere una buona precisione
Criteria crit = new Criteria();
crit.setAccuracy(Criteria.ACCURACY_FINE);
// Ed otteniamo l'ultima posizione valida dal miglior LocationProvider identificato
String provider = lm.getBestProvider(crit, true);
Location loc = lm.getLastKnownLocation(provider); // loc potrebbe essere null
```

7.10 Location API

Cambi di posizione:

```
// Acquisiamo il servizio di sistema per localizzarsi
LocationManager lm = (LocationManager) getSystemService
                                (Context.LOCATION_SERVICE);
// Definiamo un ascoltatore per i cambiamenti di posizione
LocationListener listener = new LocationListener(){
      public void onLocationChanged(Location location){
            Log.d(TAG, "Nuova posizione rilevata");
      }
      ...
}
```

7.10 Location API

Cambi di posizione:

```
LocationListener listener = new LocationListener(){
        ...
        public void onStatusChanged(String provider, int status, Bundle extras){}
        public void onProviderEnabled(String provider){}
        public void onProviderDisabled(String provider){}
};
// Registriamo l'ascoltatore al LocationManager per ricevere aggiornamenti
lm.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0, 0, listener);
```

7.10 Location API

Cambi di posizione:

public void requestLocationUpdates(String ***provider***, long ***minTime***, float ***minDistance***, LocationListener ***listener***);

provider = a chi affidarsi per la gestione della posizione

minTime = l'intervallo minimo di tempo tra due posizioni notificate

minDistance = la distanza minima espressa in metri tra due posizioni notificate

listener = l'oggetto che verrà notificato degli eventi

7.10 Location API

Posizione attuale e cambi di posizione:

un problema per la privacy!

È necessario specificare i permessi nel manifest

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

7.10 Location API

Forward geocoding

```
List<Address> indirizzi = null;
// Dato un indirizzo ottiene latitudine e longitudine relative ad esso
indirizzi = new Geocoder(this).getFromLocationName("Viale Dante, Riccione", 1);
Log.d(TAG, "Viale Dante a latitudine " + indirizzi.get(0).getLatitude() +
                    " longitudine " + indirizzi.get(0).getLongitude());
```

7.10 Location API

Reverse geocoding

```
List<Address> indirizzi = null;
// Dato un indirizzo ottiene latitudine e longitudine relative ad esso
indirizzi = new Geocoder(this).getFromLocation(43.907505, 12.911167, 1);
if(indirizzi.size() > 0){
        Log.d(TAG, "Indirizzo a latitudine" + indirizzi.get(0).getLatitude() +
                " longitudine " + indirizzi.get(0).getLongitude() + ": " +
                indirizzi.get(0).getLocality() + ", " +
                indirizzi.get(0).getCountryName());
}
```