

LAB 6

Data storage, Immagini



Goals

- Shared Preferences
- Salvataggio su File
- Android Drawable
- Creare icone personalizzate



Shared preferences

Permettono il salvataggio di una collezione di coppie chiave-valore.

Gli oggetti `SharedPreferences` puntano ad un file che contiene semplici coppie chiave-valore e offrono metodi per leggere e scrivere queste coppie.

Posso avere un file di Preferences condiviso con altri componenti dell'applicazione oppure averne uno dedicato ad una Activity.

Posso recuperare un oggetto di tipo `SharedPreferences` chiamando i seguenti metodi:

`getSharedPreferences()` — nel caso siate interessati ad avere più file di preferences condivisi da qualsiasi punto della vostra applicazione e identificati attraverso un nome univoco. Questo metodo può essere chiamato da qualsiasi classe che estenda `Context`.

`getPreferences()` : da usare nel caso in cui vogliate avere un file di preferences dedicato all'activity e visibile solo da essa. Essendo riferito all'activity stessa, non è necessario specificarne un nome univoco.

N.B nel caso in cui il file di Preferences non esista nel momento della chiamata `getSharedPreferences()` o `getPreferences()`, verrà creato nel momento in cui si cercherà di recuperare un Editor (spiegato nel dettaglio nelle successive slide)



Scrivere e leggere preferences

Per scrivere in un file di Shared Preferences è necessario recuperare un oggetto `SharedPreferences.Editor` chiamando il metodo `edit()` sull'oggetto `SharedPreferences`. Dopodiché ottenuto un editor è possibile scrivere preferences attraverso i metodi `putInt()`, `putString()` ecc..

Infine chiamare i metodi `commit()` o `apply()` per salvare i cambiamenti.

La differenza tra le due è che `commit()` essendo asincrona è bene che non sia chiamata dal main thread, in quanto metterebbe in pausa il rendering della UI.

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putInt(getString(R.string.saved_high_score_key), newHighScore);
editor.commit();
```

Per recuperare i dati salvati nelle preferences chiamare i rispettivi metodi `getInt()`, `getString()` che offrono anche la possibilità di specificare tra i parametri in ingresso un valore di default ritornato nel caso in cui la chiave specificata non sia presente.

```
int highScore = sharedPref.getInt(getString(R.string.saved_high_score_key), defaultValue);
```



Files

Android ha due aree dedicate allo storage di file:

- **Internal:** sempre disponibile è una memoria interna del sistema, in cui le app possono scrivere file privatamente.
- **External:** sono tutte le memorie esterne come USB, SD Card, tra queste rientra anche una porzione della memoria interna che viene “emulata” come SD.

Quale memoria usare?

Internal – Quando i file devono essere solo letti o scritti dall’app stessa, senza che nessun’altra possa accedervi. Nota importante: Abbiamo certezza che questa memoria è sempre presente, inoltre quando l’utente disinstalla l’app il sistema operativo rimuove i file salvati dall’app dalla memoria interna

External – Quando vogliamo che altre applicazioni possano sia leggere che modificare i file scritti dalla nostra applicazione. Inoltre i file presenti nella memoria esterna sono “visibili” anche quando colleghiamo lo smartphone al PC. Nota importante: la memoria esterna “emulata” è tipicamente presente in ogni smartphone, mentre memorie esterne come USB o SD possono essere rimosse.



Scrivere e leggere su memoria interna

Per scrivere e leggere un file si può utilizzare la classica metodologia vista in generale su Java. Esiste il metodo “*getFilesDir()*”, che restituisce un “File” che rappresenta la directory corretta sulla memoria interna, sul quale si può leggere e scrivere. (scrivilo in corsivo per fare capire che è un metodo)

```
private void writeInternalFileOld(final String fileName, final String text, boolean append) throws IOException {  
    final File file = new File(this.getFilesDir(), fileName);  
    final FileOutputStream outputStream = new FileOutputStream(file, append);  
    outputStream.write(text.getBytes());  
    outputStream.close();  
}
```

In alternativa Android mette a disposizione un metodo “*openFileOutput(String filename, int flags)*” che restituisce direttamente un *FileOutputStream*. Esiste un metodo equivalente per l’accesso in lettura “*openFileInput(String filename)*”.

```
private void writeInternalFile(final String fileName, final String text, final boolean append) throws IOException {  
    int flags = Context.MODE_PRIVATE;  
    if(append) flags = flags | Context.MODE_APPEND;  
    final FileOutputStream outputStream = openFileOutput(fileName, flags);  
    outputStream.write(text.getBytes());  
    outputStream.close();  
}
```

Flags:

- *Context.MODE_PRIVATE* = utilizzato di default, il file creato con questo flag è privato e nessuna app esterna può accedervi.
- *Context.MODE_APPEND* = permette di scrivere su file in modalità “append”.



Scrivere e leggere su memoria esterna

In questo caso Android non mette a disposizione metodi particolari per lettura e scrittura, quindi è necessario utilizzare le classiche metodologie di Java, quindi `BufferedReader`, ecc... Esistono differenti directory esterne in cui si possono salvare i file:

- “`Environment.getExternalStorageDirectory()`”: root directory della memoria SD.

```
final File file = new File(Environment.getExternalStorageDirectory(), "nomefile.txt");
```

- “`getExternalFilesDirs()`”: directory specifica dell’applicazione nella memoria SD (`Android/com.example.MyApplication/files/`)

```
final File file = new File(getExternalFilesDir( type: null), fileName);
```

- “`Environment.getExternalStoragePublicDirectory(String type)`” : directory esterna che varia a seconda del tipo di file da salvare, solitamente si utilizza per memorizzare foto, documenti o download, infatti il parametro “type” specifica la tipologia di file da memorizzare (es. `Environment.DIRECTORY_DOWNLOADS`).

```
final File file = new File(Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS), "nomefile.docx");
```



- In generale, visto che le memorie esterne come USB o SD possono essere rimosse è necessario verificare, prima di effettuare una scrittura o una lettura, lo stato della memoria esterna.

```
public boolean isExternalStorageReadable() {  
    String state = Environment.getExternalStorageState();  
    if (Environment.MEDIA_MOUNTED.equals(state) ||  
        Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {  
        return true;  
    }  
    return false;  
}
```

```
public boolean isExternalStorageWritable() {  
    String state = Environment.getExternalStorageState();  
    if (Environment.MEDIA_MOUNTED.equals(state)) {  
        return true;  
    }  
    return false;  
}
```



Accenno ai permessi

Le applicazioni Android per poter accedere a componenti particolari, o a dati dell'utente necessitano di permessi. L'elenco di tutti i permessi che un'applicazione necessita devono essere dichiarati nel file manifest dell'applicazione. **Nel caso specifico dei file, la lettura e scrittura su memoria interna non necessita di permessi speciali, cosa che invece richiede la memoria esterna.**

Nel manifest è sufficiente aggiungere due permessi.

```
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />  
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

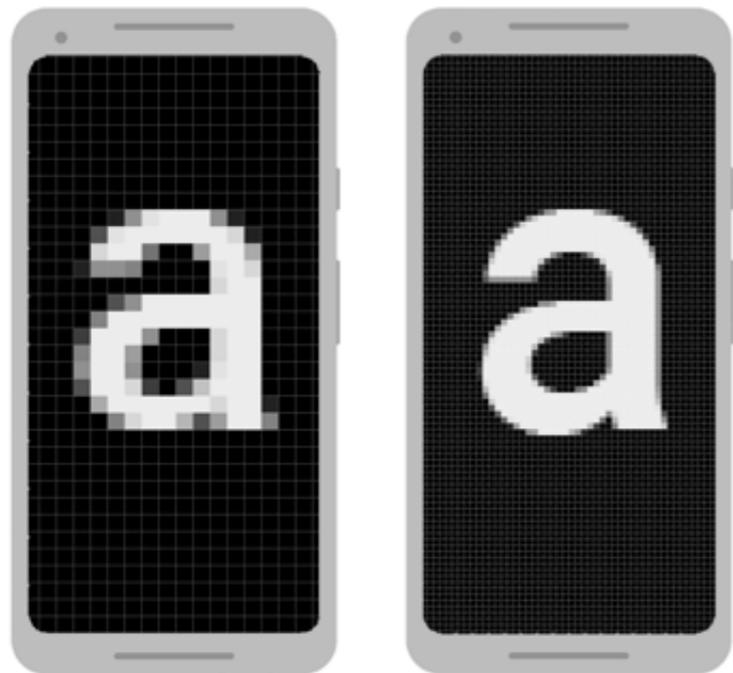


Supportare differenti dimensioni degli schermi

La regola di base per essere indipendente dalla risoluzione è evitare di usare l'unità di misura px (pixel) per definire distanze e misure.

Usare l'unità di misura in pixel è un problema perché schermi differenti hanno una densità di pixel differenti, e lo stesso numero di pixel può corrispondere a grandezze differenti grandezze fisiche in dispositivi diversi.

Si preferisce quindi l'utilizzo dell'unità di misura *density-independent pixels* (dp) come unità di misura Android traduce automaticamente questo valore nell'appropriato numero di pixel.



L'immagine a fianco mostra due schermi con stessa dimensione ma diverso numero di pixel.

Qual'è l'impatto di uso di px sull'una rispetto all'altra?

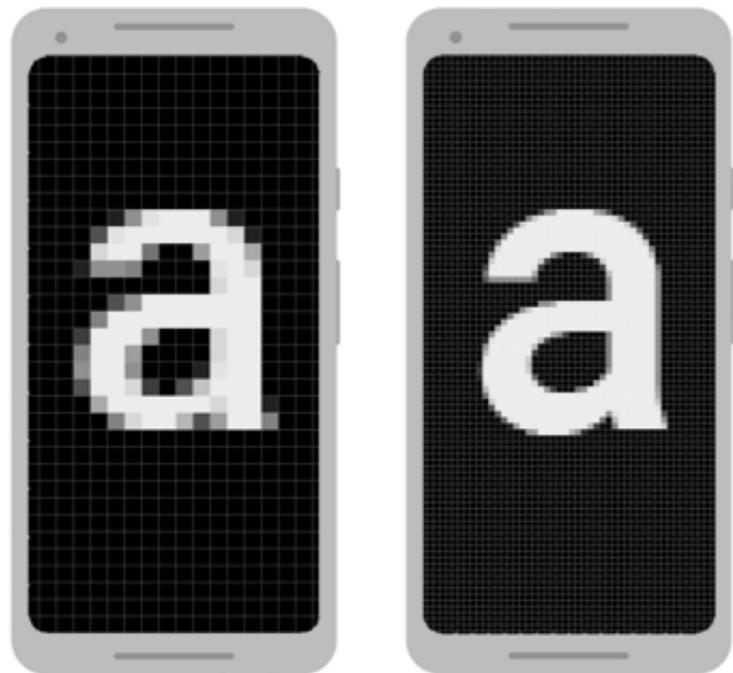


Supportare differenti dimensioni degli schermi

La regola di base per essere indipendente dalla risoluzione è evitare di usare l'unità di misura px (pixel) per definire distanze e misure.

Usare l'unità di misura in pixel è un problema perché schermi differenti hanno una densità di pixel differenti, e lo stesso numero di pixel può corrispondere a grandezze differenti grandezze fisiche in dispositivi diversi.

Si preferisce quindi l'utilizzo dell'unità di misura *density-independent pixels* (dp) come unità di misura Android traduce automaticamente questo valore nell'appropriato numero di pixel.



Apparirà più grande nel device di sinistra!!

https://developer.android.com/training/multiscreen/screendensities.html#use_vector_graphics_instead

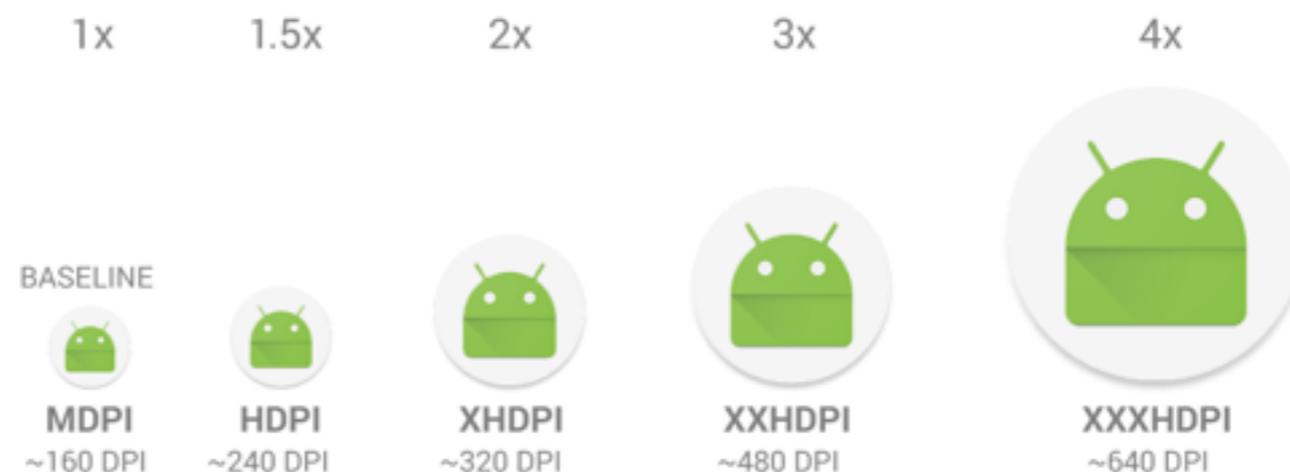


Drawable, mipmap per schermi differenti

Un discorso a parte meritano le immagini. I modificatori applicati alle cartelle drawable (ldpi, mdpi, hdpi e via dicendo) sono alcune delle sigle che identificano le densità dei display.

Sarà il sistema che selezionerà l'immagine appropriata basandosi sulla densità dello schermo ogniqualvolta farete `@drawable/awesome-image`

```
res/  
drawable-xxxhdpi/  
  awesome-image.png  
drawable-xxhdpi/  
  awesome-image.png  
drawable-xhdpi/  
  awesome-image.png  
drawable-hdpi/  
  awesome-image.png  
drawable-mdpi/  
  awesome-image.png
```



Stesso discorso vale per le mipmap, utilizzate per l'icona della vostra applicazione.



Immagini vettoriali

Una delle novità introdotte da Android 5.0 è costituita dal supporto per le immagini vettoriali; una immagine vettoriale è costituita non da un insieme di colori (uno per ogni pixel) , ma da un insieme di curve da disegnare.

Proprio per questa ragione una immagine vettoriale può essere scalata senza perdita di qualità.

Lo standard SVG (scalable vector graphics) consente, tramite la definizione di un **file xml** , di definire le curve da disegnare per disegnare un'immagine.

Il file xml finirà nella cartella res/drawable.

Dal momento in cui una immagine vettoriale può essere usata per tutte le densità di pixel il file xml viene inserito nella directory drawable di default.



Best practices

- Utilizzare il più possibile i RelativeLayout, che ha per sua natura si adatta ai display;
- Nel dimensionamento di View e Layout, evitare di dichiarare esplicitamente le misure assolute , ricorrendo invece il più possibile ai valori wrap_content e match_parent;
- Tutte le misure necessarie vanno indicate in dp (o in sp per i font) evitando qualsiasi altra unità di misura che abbia attinenza col mondo reale (pixel, millimetri, pollici, etc...);
- Diversificare le risorse in base alle possibili configurazioni dei dispositivi, predisponendo i nomi delle cartelle caratterizzati dagli appositi modificatori. (directory drawable-ldpi, drawable-hdpi)



Glide

Molto spesso ci si trova a dover visualizzare immagini provenienti da fonti esterne alle risorse dell'app (Immagini provenienti dal Web).

- Gestire in modo asincrono il caricamento delle immagini può essere un task di difficile gestione e un facile generatore di bug.

Soluzione:

Glide è una libreria open source scritta in Java per piattaforma Android che si occupa di una sola cosa: caricare immagini in modo efficiente.

Per utilizzare Glide all'interno di un progetto operare nel seguente modo:

1. includere la libreria nel file `.gradle/app` nella sezione `dependencies`
2. Dichiarare nel file `manifest` il permesso di utilizzare la connessione internet
3. Utilizzare la classe `Glide` per caricare in modo asincrono immagini e visualizzarle in un componente grafico di tipo `ImageView`

<https://android.jlelse.eu/best-strategy-to-load-images-using-glide-image-loading-library-for-android-e2b6ba9f75b2>



1

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.android.support:appcompat-v7:28.0.0'
    implementation 'com.android.support.constraint:constraint-layout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.2'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'
    //Glide
    implementation 'com.github.bumptech.glide:glide:4.9.0'
    annotationProcessor 'com.github.bumptech.glide:compiler:4.9.0'
}
```

2

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.lab_06">
    <uses-permission android:name="android.permission.INTERNET" />
```

3

```
Glide.with( activity: this).
    load( string: "https://anyurlimage...")
    .into(imgView);
```



Esercizio

L'esercitazione odierna è volta all'utilizzo delle Shared Preferences per permettere la multi-utenza.

Si utilizzino le shared Preferences per memorizzare l'utente loggato. Per ogni utente che effettuerà login si avrà quindi una lista di contatti personalizzata.

Aggiungere alla precedente esercitazione:

- **Una schermata per il login**
- **Il salvataggio di una lista di contatti per ogni utente che si è loggato fino ad ora**
- **Un menu nell'action bar che permetta il logout in qualsiasi momento**
- **L'aggiunta di una immagine come logo nella pagina di login**
- **L'aggiunta di una icona per l'applicazione**