Sviluppo di Mobile Software

Android

Indice argomenti

CAPITOLO 1 - Introduzione ad Android

Cos'è Android?

Android non è né un linguaggio di programmazione né un browser, ma un vero e proprio **stack** che comprende componenti che vanno dal sistema operativo fino a una virtual machine per l'esecuzione delle applicazioni. La sua caratteristica fondamentale è l'utilizzo di **tecnologie open source** a partire dal sistema operativo, che è Linux.
Si tratta, infatti, di un **sistema aperto** poiché permette il controllo su qualsiasi cosa, anche sul linguaggio che si utilizza per assemblare/buildare le app (*fase di creazione dell'APK*).
Android viene aggiornato almeno una volta l'anno. Ogni versione successiva è (quasi) pienamente compatibile con le precedenti. I cambiamenti nelle API sono identificati da un **API Level**. Infatti, le applicazioni possono dichiarare:

- Un **API Level minimo** di cui necessitano per funzionare (un'applicazione realizzata per un valore di API Level pari a 7 potrà essere eseguita senza problemi su dispositivi che utilizzano una versione superiore)
- Un **API Level target** per cui sono state scritte
- Un **API Level massimo** oltre il quale non funzionano più (sconsigliato e obsoleto).

Nel debug, l'uso di quale classe ci permette di visualizzare messaggi sulla console?

Il **debugger** è integrato in Android Studio ed è utile per analizzare lo stato dell'applicazione tramite i breakpoints. Viene utilizzato soprattutto con poche linee di codice o per problemi specifici, mentre per problemi più complessi c'è il **logcat** con comandi ed etichette di messaggi.
La **classe Log** dispone di un certo insieme di metodi statici per la visualizzazione di messaggi di log associati alle classiche priorità.
Per stampare un messaggio di log si usa la seguente sintassi:

Log.w(TAG, MESSAGE);

dove, **TAG** è l'etichetta che ci permette di individuare il log all'interno del logcat, mentre **MESSAGE** è il messaggio da visualizzare.
Le tipologie di log sono:

- Log.e(String, String) - **error**
- Log.w(String, String) - **warning**
- Log.i(String, String) - **information**
- Log.d(String, String) - **debug**
- Log.v(String, String) - **verbose** (senza tipo).

Dire a cosa serve utilizzare un emulatore, con quale strumento comunicare con esso e quali sono i suoi limiti.

Un emulatore serve per creare uno o più dispositivi virtuali, assegnando loro un nome e specificando le caratteristiche di ognuno, in maniera tale da testare le applicazioni evitando l'utilizzo di un dispositivo reale.

Android Studio permette di emulare diversi dispositivi e configurarli con diversi SDK attraverso l'Android Virtual Device Manager (AVD). Dall'AVD è, infatti, possibile impostare la velocità della rete, lo stato della batteria, l'orientamento, etc.
E' possibile comunicare con l'emulatore, tramite il terminale di Android Studio, dove da riga di comando è possibile inviare comandi ed innescare eventi.
Con esso si comunica tramite **Android Virtual Device Manager** (AVD Manager), un'interfaccia esposta da Android Studio. D'altra parte, l'emulatore essendo virtuale, presenta dei **limiti**, vale a dire non supporta tutte le caratteristiche di un dispositivo come i sensori, il bluetooth, etc.. Altri limiti sono la velocità e le prestazioni.

Cos'è un APK?

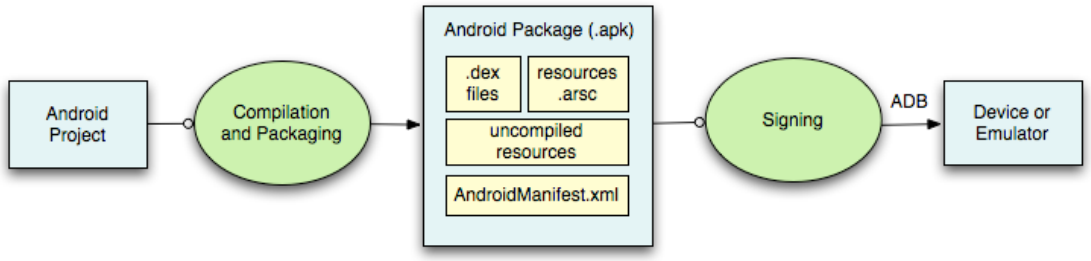
APK (**Android Package Kit**) è il formato standard di file utilizzato per distribuire e installare applicazioni Android. Contiene tutti i file necessari per eseguire un'app su un dispositivo.
Si tratta, dunque, di un package che si ottiene come risultato della build di un progetto e contiene:

- Un file classes.dex (Dalvik Executable), il quale contiene il bytecode risultante dalle classi del codice sorgente java
- Un file resources.arsc che contiene le risorse compilate
- La cartella res che contiene le risorse non compilate (stringhe, immagini, etc.)
- Il file AndroidManifest.xml.

Ogni APK ha sicuramente una firma di debug o di release che identifica lo sviluppatore.
I requisiti per l'installazione su un dispositivo sono:

1. il pacchetto deve essere integro
2. l'APK deve supportare la versione di Android in uso sullo smartphone
3. se il dispositivo possiede le funzionalità dichiarate nei tag <uses-feature> (in caso di required = 'true')
4. se il certificato con cui è stata firmata l'APK è ancora valido.

CAPITOLO 2 - La firma di un APK



Quali sono le differenze tra firma in modalità debug e in modalità release.

Android richiede che tutti gli APK siano firmati digitalmente prima di essere installati su un dispositivo o aggiornati. La **firma di un'app Android** è un processo critico per garantire l'integrità e l'autenticità del codice dell'applicazione. A livello teorico, il processo di firma

utilizza crittografia asimmetrica per associare un'applicazione a un'identità specifica (di solito quella dello sviluppatore) e per proteggere l'applicazione da modifiche non autorizzate.
Per i test è bene firmare l'app in **modalità debug**, in questo modo è utilizzata una chiave con password nota, per cui non è necessario inserirla per far partire l'app. Gli Android Development Tools firmano l'apk e si occupano di tutto purché lo sviluppatore **fornisca un certificato valido**. La prima volta che si esegue l'app o si lancia il debug del progetto in Android Studio, l'IDE crea automaticamente il keystore di debug e il certificato in $HOME/.android/debug.keystore che ha una chiave fissa con password predefinita.
Invece, in **release mode** è necessario firmare l'app esplicitamente, in particolare se sarà caricata nel Play Store. Richiede un keystore personalizzato creato dallo sviluppatore, contenente una chiave privata. Se si perde il keystore, si perde anche la possibilità di aggiornare l'app.

Quali passi bisogna seguire per ottenere un certificato valido per firmare un apk?

Per ottenere un **certificato valido** occorre seguire questi passaggi:

1. Si crea un **keystore**, cioè un file binario che contiene le chiavi private, da conservare in un posto sicuro
2. Si crea una chiave privata, che rappresenta l'entità associata all'app (azienda/persona). E' possibile generare un keystore e una chiave per firmare un app direttamente da Android Studio, impostando il percorso del keystore, l'alias e la password della nuova chiave privata e le informazioni sul certificato da associare alla chiave.
3. Si aggiunge la configurazione per la firma al modulo corrente dell'app da firmare (il predefinito è app)
4. È invocato il build task assembleRelease di Android Studio
5. Il package app/build/apk/app-release.apk è firmato.

```
...
android {
    ...
    defaultConfig { ... }
    signingConfigs {
        release {
            storeFile file("myreleasekey.keystore")
            storePassword "password"
            keyAlias "MyReleaseKey"
            keyPassword "password"
        }
    }
    buildTypes {
        release {
            ...
            signingConfig signingConfigs.release
        }
    }
}
...
```

Questo frammento di codice rappresenta la firma associata alla release in cui vengono riportati i parametri (definiti dallo sviluppatore) che vanno dall'apertura della parentesi graffa affianco a 'release' fino al primo rigo della chiusura della prima delle due parentesi al di sopra di buildTypes.

Tuttavia, non è una buona idea inserire la password dello storechain e la password dell'app nel file di build, meglio fare in modo che siano prese da variabili d'ambiente oppure fare in modo che queste vengano chieste all'utente.

Il **certificato di debug** dura 30 anni dopodiché si avrà un errore di build, in tal caso occorre eliminare il file debug.keystore e creare un nuovo certificato.

E' importante non perdere il keystore generato poiché quando il sistema installa un aggiornamento per l'app, confronta i certificati della nuova versione con quelli della versione esistente. Il sistema consente l'aggiornamento se i certificati corrispondono. Pertanto se il certificato viene perso e si firma la nuova versione con un certificato diverso, è necessario assegnare un nome pacchetto diverso all'app: in questo caso, l'utente installa la nuova versione come app completamente nuova.

Schema di firma V2

Introdotto in Android Nougat (7.0), garantisce un'installazione più veloce e maggiore protezione da alterazioni di apk. Inoltre, se si firma con APK Signature Scheme v2 e poi si apportano modifiche all'app, la firma viene invalidata.

Per **disabilitare** lo schema V2 e firmare in modo tradizionale è sufficiente aprire build.gradle e impostare la proprietà v2SigningEnabled a 'false'.

Perdere la password di un'app pubblicata nello store comporta l'impossibilità di aggiornare l'app, perché ad ogni aggiornamento bisogna firmare l'app.

Da Agosto 2021 per pubblicare una nuova app su Google Play si deve:

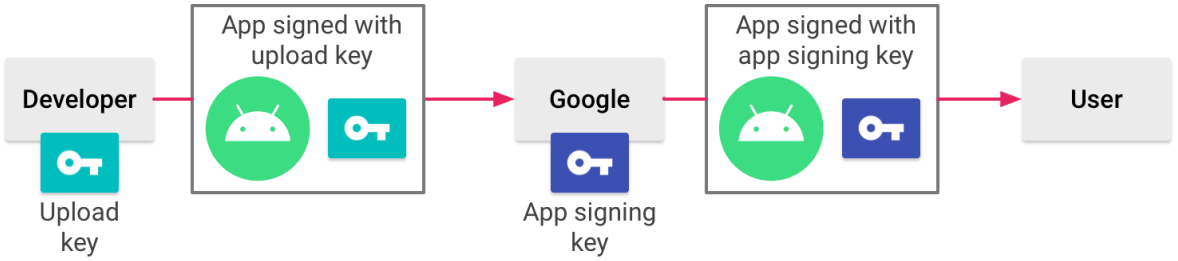
1. Generare una **upload key** e un keystore (simile alla firma di release)
2. Firmare la app con la upload key (simile al funzionamento della release key)
3. Configurare il servizio **Play App Signing** con un'app signing key
4. Caricare la app sul Google Play console
5. Preparare e configurare la release finale su Google Play in modo che sia verificata e firmata da Google.

Con Play App Signing, Google gestisce e protegge la chiave di firma e le utilizza per firmare le APK per la distribuzione.

Play App Signing utilizza due chiavi:

1. **App signing key** o **chiave di firma**: è parte del 'modello di aggiornamento sicuro' di Android, non cambia durante la vita dell'app e può essere fornita dall'autore o generata da Google
2. **Upload key**: chiave usata per firmare la app prima del caricamento sul Play Store, è privata ma di proprietà dell'autore e può essere la release key di Android Studio.

Google utilizza il certificato di upload per verificare l'identità dell'autore e poi firma l'APK con la app signing key di distribuzione sul suo store.



Utilizzando una chiave di caricamento separata è possibile richiedere un ripristino della upload key se viene smarrita o rubata (in confronto, per le app create prima di agosto 2021

che non hanno attivato la firma dell'app, perdendo la chiave di firma si perde la possibilità di aggiornare l'app).
Le chiavi sono archiviate nella stessa infrastruttura utilizzata da Google per archiviare le proprie chiavi.
Grazie a Play App Signing, se si perde la chiave di caricamento o se è compromessa si può contattare Google per revocarla e generarne una nuova. Poiché la chiave di firma dell'app è protetta da Google, è possibile continuare a caricare nuove versioni dell'app come aggiornamento dell'originale, anche se si modifica la chiave di caricamento.

Come preparare l'app per il deploy?

Per distribuire un'app Android, è necessario firmare l'APK o l'App Bundle con una chiave privata. Dunque, occorre:

1. generare un keystore con una chiave privata, se non presente
2. firmare l'APK, selezionando il Build Type 'Release' e il tipo di firma da applicare (V1, V2 o entrambe). Nel file build.gradle avremo:

```
buildTypes {
    release {
        minifyEnabled true
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
'proguard-rules.pro'
    }
}
```

3. avviare il processo di build.

Al termine, l'APK o l'App Bundle verrà generato nella cartella release del progetto.
Minified se impostato a true, rimuove risorse e codice del progetto che non sono utilizzate nell'APK finale, in modo da renderlo più piccolo possibile.
ProGuard è un tool integrato in Android Studio che ottimizza il codice eliminando elementi inutilizzati e offuscandolo per ridurre il rischio di decompilazione.

CAPITOLO 3 - L'architettura Android

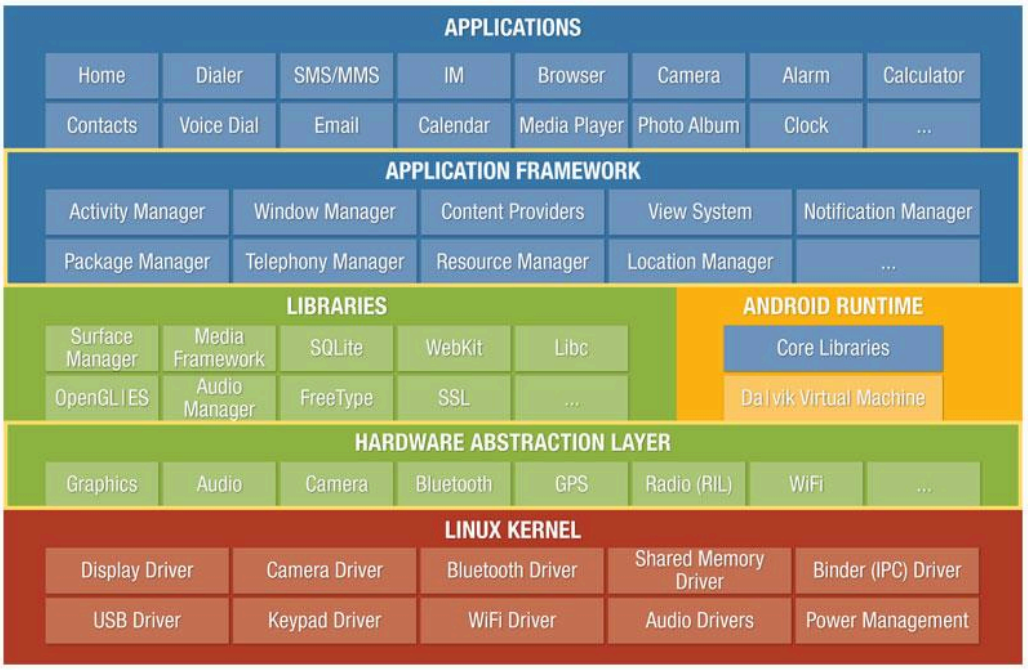
Illustrare scopi e componenti dell'Open Headset Alliance.

L'**OHA**, assieme a Google, costituisce un consorzio di aziende con l'obiettivo di costruire un **ecosistema** di produzione e sviluppo (HW e SW) aperto e compatibile: **Android**, un ambiente evoluto per la realizzazione di applicazioni per dispositivi mobili studiato da più di 84 aziende (eterogenee e con diversi obiettivi, interessi e culture) che si uniscono per accelerare l'innovazione e offrire ai clienti un prodotto economico ma tecnologicamente avanzato, fornendo una User Experiences moderna e adatta alle varie tipologie di utenti, bisogni e culture.
I membri comprendono: **operatori mobili** (Vodafone, Telecom Italia), **produttori di semiconduttori** (Intel Corporation, NVIDIA), **produttori di dispositivi** (Acer, LG, Samsung), **produttori di software** (eBay, Google) e **compagnie di commercializzazione**.
Far collaborare tutte queste aziende fra loro, con interessi e obiettivi diversi non è semplice. Infatti molti competitors sono praticamente scomparsi o si sono focalizzati su un determinato

mercato (Nokia), non sono stati capaci di creare questa alleanza. Questo per sottolineare l'importanza della negoziazione e della collaborazione.

Illustrare i livelli dell'architettura Android, specificando il ruolo che tali rivestono.

L'architettura di Android è descritta come una **pila** (stack), ossia i componenti principali sono organizzati secondo una struttura a **layer**. Un'architettura a layer permette di fare in modo che ciascuno strato utilizzi i servizi dello strato sottostante per fornire allo strato superiore altri servizi, di più alto livello.



Linux kernel: è il livello più basso, per i costruttori di Hardware, componenti elettronici e segnali, mette a disposizione i servizi core. Contiene l'implementazione di una serie di driver di interazione con l'HW (display driver, camera driver, bluetooth driver) i quali permettono il funzionamento delle periferiche HW e la comunicazione con il sistema operativo.
Dunque, questo layer fornisce i servizi principali di un sistema operativo:

- sicurezza (permessi)
- gestione dei processi e comunicazioni
- gestione e condivisione della memoria
- gestione file e I/O network
- drivers e dispositivi (accesso ai sensori)
- componenti specifici Android (gestione consumi e batteria).

Hardware abstraction layer (HAL): è un livello software che serve a fare da ponte tra il sistema operativo Android e l'hardware fisico del dispositivo. Permette al sistema Android di interagire con i componenti hardware (come fotocamera, microfono, sensori, processore, ecc.) senza dover gestire direttamente le complessità specifiche di ogni dispositivo o produttore, garantendo, in questo modo, che Android funzioni su una vasta gamma di dispositivi con HW diverso.
Libraries: librerie native scritte in C/C++ (attraverso il Native Development Kit) che espongono interfacce in java per l'interazione con i classici servizi di un dispositivo mobile,

come la persistenza dei dati, la grafica, la gestione dei font, etc. Inoltre, le interfacce Java vengono utilizzate da un altro componente fondamentale che si chiama Core Libraries.
Android Runtime: le applicazioni utilizzano le API messe a disposizione dalle Core Libraries per accedere ai servizi implementati dal layer delle Libraries. Il tutto viene poi eseguito dal componente chiamato ART. Si tratta del runtime di android che scrive ed esegue app Android ed è composto dalle Core Java Libraries, Dalvik VM e ART.
Application Framework: utilizza i servizi sia del Runtime che delle Libraries e contiene una serie di componenti di alto livello utili alla realizzazione di tutte le applicazioni Android. Contiene software riusabile e utile in molte app: servizi e librerie che possiamo includere all'interno dell'app. **Es.** il View System contiene i comuni elementi grafici (pulsanti, icone, etc.).
Vediamo singolarmente i componenti che costituiscono questo layer:

- **Package Manager:** database che tiene traccia delle app installate nel dispositivo, consente ad un'app di trovare, mettersi in comunicazione e richiedere i servizi ad un'altra app (sfrutta il meccanismo delle intent).
- **Window Manager:** gestisce le finestre che compongono un'app (es. notification bar, main application, etc.)
- **View System:** gestisce elementi dell'UI (icone, testo, pulsanti, etc.)
- **Resource Manager (oggetto R in Java):** gestisce le risorse non compilate come stringhe, elementi grafici, file di layout. Es. le stringhe in inglese possono essere tradotte in italiano cambiando le impostazioni.
- **Activity Manager:** gestisce il ciclo di vita delle app e lo stack per la navigazione.
- **Content Provider:** database che consente alle app di memorizzare e condividere le informazioni. Es. l'app del telefono accede alle informazioni dei contatti e li usa per effettuare le telefonate, ma anche l'app per gli SMS può usare i contatti.
- **Location Manager:** permette alle app di ricevere informazioni sul movimento e relative alla posizione (GPS) del dispositivo. Consente di svolgere task specifici come trovare la posizione e la direzione.
- **Notification Manager:** consente alle app di inserire informazioni nella notification bar per esempio per avvisare l'utente che si è verificato uno specifico evento.

Application: si riferisce a tutte le applicazioni installate nel dispositivo. Nessuna di queste app è parte del sistema, se si ha un app preferita la si può installare al posto di qualsiasi app standard.

Spiegare il ruolo del Package Manager nell'architettura Android.

Il **Package Manager è un database** che tiene traccia delle app installate nel dispositivo e consente ad un'app di trovare, mettersi in comunicazione e richiedere servizi ad un'altra app. Si può utilizzare la classe PackageManager per il recupero di vari tipi di informazioni relative ai pacchetti dell'applicazione attualmente installati sul dispositivo. Puoi trovare questa classe attraverso il metodo **getPackageManager()**.

Illustrare le differenze principali esistenti tra Dalvik e ART. Spiegare, inoltre, in che contesto si parla di ahead-of-time e perché è stata adottata tale tecnologia.

Android fa uso di una macchina virtuale come ambiente runtime per eseguire i file APK che costituiscono un'applicazione. Difatti i file APK non sono altro che package in cui le classi del

codice sorgente java sono state compilate in un file DEX (Dalvik Executable), che includono il bytecode, mentre le risorse non compilate e il manifest rimangono tali.
Fino ad **Android 4.4 KitKat**, la macchina virtuale utilizzata era **Dalvik**, basata sulla tecnologia **JIT (Just-In-Time)**. In questo sistema, l'applicazione veniva parzialmente compilata dallo sviluppatore tramite l'IDE, mentre Dalvik si occupava di convertire il codice in linguaggio macchina in tempo reale durante l'esecuzione (interprete). Questo approccio migliorava le prestazioni rispetto al Java nativo, poiché la parte compilata in fase di sviluppo era ottimizzata per il codice, mentre quella elaborata da Dalvik veniva adattata dinamicamente al dispositivo su cui l'app era in esecuzione.
Da **Android 5.0** in poi Dalvik è stato sostituito con **ART**, un ambiente di runtime che utilizza la tecnologia **AOT (ahead-of-time)**, cioè al momento dell'installazione, compila il bytecode utilizzando lo strumento **dex2oat** sul dispositivo. Questa utility accetta i file DEX come input e genera un prodotto finale specifico per il dispositivo.
Le differenze sono che Dalvik, usando JIT, garantisce un minor consumo di spazio di archiviazione, ma tempi di esecuzione maggiori, pertanto è adatta per i dispositivi con uno storage interno poco ampio. Mentre ART, per via di AOT consuma molto più spazio di archiviazione interno in quanto memorizza le applicazioni compilate in aggiunta alle APK, ma l'esecuzione è molto più veloce poiché la compilazione viene eseguita solo in fase di installazione (che durerà qualche secondo in più) e c'è un minor utilizzo del processore.

CAPITOLO 4 - Android Manifest, gradle e risorse

Qual è il ruolo svolto dall'Android Manifest

L'Android Manifest è il file di configurazione principale di un'app Android. Fornisce al sistema operativo e al Play Store tutte le informazioni essenziali sull'app, tra cui:

- **Identità dell'app:** nome del package e versione.
- Specifica la **versione minima** e **target** di Android supportare, tramite il tag **<uses-sdk/>**
- Può definire la compatibilità con determinate caratteristiche hardware <*uses-feature* android:name="android.hardware.camera" *android:required="true"*/>
- **Componenti principali:** Activity, Service, BroadcastReceiver e ContentProvider, con dettagli sul loro funzionamento e interazione.
- **Permessi:** richieste di accesso a risorse sensibili (fotocamera, posizione, ecc.). Es. <uses-permission android:name="android.permission.CAMERA"/>
- **Configurazione dell'interfaccia:** orientamento dello schermo, icone, intent filter e Activity di avvio.
- **Altri parametri avanzati:** processi ospitati dalle componenti e classi di Instrumentation per i test.

In sintesi, il Manifest definisce il comportamento dell'app e la sua integrazione con il sistema Android.

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="it.uniba.di.ivu.sms16.myapp">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="MiaApp"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Spiegare il ruolo di Gradle nella realizzazione delle App Android.

Gradle è un tool progettato per gestire la compilazione e il packaging delle applicazioni, ottimizzato per ambienti multi-progetto e approcci incrementali. Una delle sue caratteristiche principali è la capacità di determinare automaticamente quali parti del progetto necessitano di essere aggiornate.
Gradle utilizza un Domain Specific Language (DSL), basato su **Groovy**, un linguaggio che gira sulla JVM (Java Virtual Machine). Grazie a questa configurazione, permette agli sviluppatori di gestire in maniera flessibile e personalizzabile l'intero processo di build delle applicazioni Android.
Ogni script di Gradle segue un processo suddiviso in tre fasi principali:

- **Inizializzazione:** Gradle legge i file di configurazione (ad esempio, build.gradle) e organizza i dati in strutture di informazioni utilizzabili.
- **Configurazione:** gli script e le variabili vengono aggiornati con i dati raccolti. Gradle stabilisce l'ordine di esecuzione dei task, tenendo conto delle dipendenze e delle sequenze di chiamate necessarie.
- **Esecuzione:** Gradle esegue i task definiti negli script. Se si verificano errori, vengono notificati immediatamente.

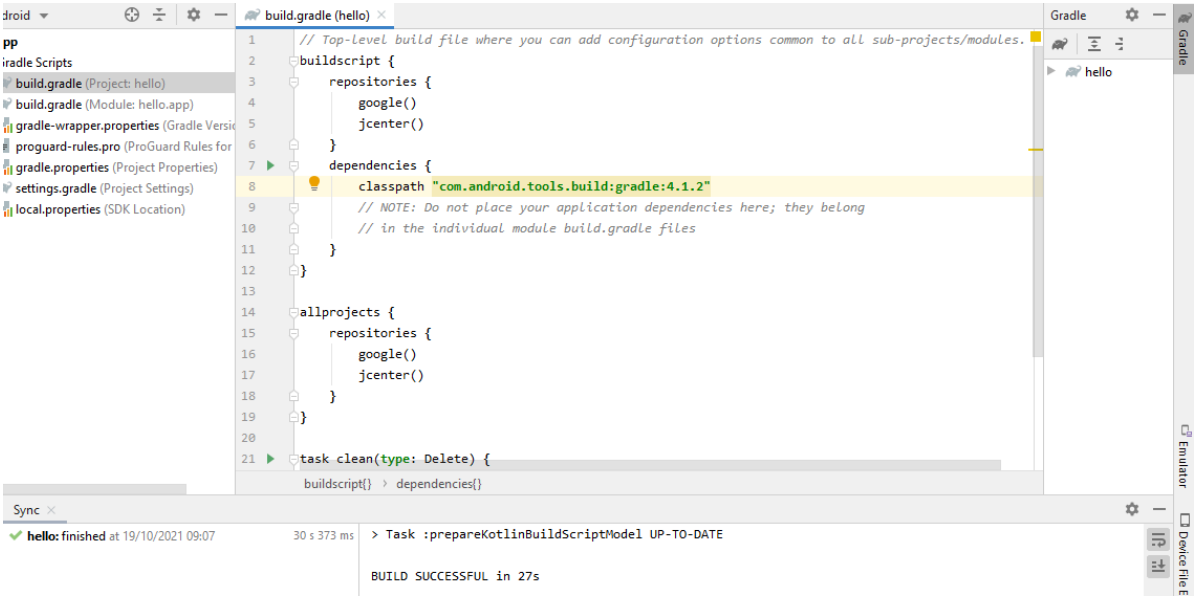
Abbiamo due file **build.gradle**, uno relativo al singolo modulo (possono essercene di più) e uno relativo al project, ossia al progetto intero.
A livello di progetto si indicano i repository delle librerie ufficiali (es. Google), cioè quelle generali. Mentre quelle legate alle funzionalità dell'app (es. database) si mettono nel build del modulo, in quanto non ha senso che siano visibili per l'intero progetto, se quella funzionalità viene utilizzata solo da alcuni moduli.
Analizziamo singolarmente i due file:
Project: contiene le impostazioni per gestire il progetto e il suo pacchetto (librerie Google, maven, jcenter, build tools, etc.).
È composto da:

- **buildscript:** serve a definire le dipendenze di build, ovvero le librerie di cui lo stesso Gradle necessita per il build dell'applicazione.
    - **repositories:** contiene le sorgenti (repository) delle dipendenze

- **dependencies:** le definizioni delle dipendenze (librerie), aggiunte al classpath. Definisce, anche, il gradle su cui viene compilata e impacchettata l'app.

- **allprojects**: informazioni relative a tutti i moduli.

Quando includiamo le librerie nel build del project, non le stiamo includendo all'interno del codice, ma stiamo impostando il puntatore al repository del sistema operativo in cui si trovano già quelle librerie dell'SDK. Dunque, fanno riferimento a quello che c'è già installato sul device. Questo permette di mantenere il codice snello.



Module: ha impostazioni per il singolo modulo dell'app. I moduli sono definiti nel file **settings.gradle**.

```
plugins {
    alias(libs.plugins.android.application)
}

android {
    namespace 'com.example.helloworldapp'
    compileSdk 34

    defaultConfig { DefaultConfig it ->
        applicationId "com.example.helloworldapp"
        minSdk 24
        targetSdk 34
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes { NamedDomainObjectContainer<BuildType> it ->
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }
    compileOptions { CompileOptions it ->
        sourceCompatibility JavaVersion.VERSION_11
        targetCompatibility JavaVersion.VERSION_11
    }
}

dependencies {

    implementation libs.appcompat
    implementation libs.material
    implementation libs.activity
    implementation libs.constraintlayout
    testImplementation libs.junit
    androidTestImplementation libs.ext.junit
    androidTestImplementation libs.espresso.core
}
```

- **compileSdk:** API di build (SDK Platform) che indica l'SDK per cui il progetto è buildato. Dunque, specifica l'API Level che andremo a considerare come effettivo per il nostro progetto.

Il **defaultConfig** contiene informazioni specifiche del modulo su cui abbiamo fatto i test: contiene le configurazioni di default dell'app le quali si andranno a fondere con quelle definite in AndroidManifest.xml.

- **applicationId:** package dell'app, è l'id nel PlayStore e fondamentale per le build variant.
- **minSdk** versione minima supportata, **compatibilità**. L'app va solo su devices con SDK >= 24
- **targetSdk:** versione sdk testata, **affidabilità**. Può coincidere con la compileSdk.
- **versionCode:** attuale versione dell'app
- **versionName:** come la precedente, ma stringa.

compileSdkVersion e buildToolsVersion sono **obbligatori**.
In genere è: *minSdk <= targetSdkVersion <= compiledSdkVersion*
In defaultConfig ci sono metadati che possono essere cambiati (build variant).

Gradle è molto utile anche nella gestione delle **dipendenze**, ovvero la dichiarazione e gestione delle librerie di cui, nelle diverse fasi di sviluppo, la nostra applicazione necessita per le proprie funzionalità.
Ci sono cinque modalità di dipendenza:

- **compile [default]:** lib aggiunta al progetto ed all'apk
- **apk:** solo nel file .apk
- **provided:** librerie non aggiunte in apk (previste come già presenti sul device). Ad es. l'app utilizza librerie già installate da una precedente nostra app (versione free e versione a pagamento)
- **testCompile ed androidTestCompile:** utilizzabili nei test.

Spiegare cosa si intende per Build Variant e come sono gestite in Android. Fornire un esempio di progettazione e configurazione.

Le build consentono di creare versioni e configurazioni diverse della stessa app. Ad esempio, è possibile sviluppare due varianti: una con immagini e un'altra senza. Queste versioni condividono codice e risorse comuni, differenziandosi solo per alcune specifiche configurazioni o contenuti.
I **build types** contengono i tipi diversi di build, dove i fondamentali sono **debug** (quello di default) e **release**. Per esempio, possiamo avere una versione 'minified' che elimina dal prodotto finale codice e risorse del progetto non utilizzate.
Gradle ha la priorità sul manifest. Pertanto, le informazioni di build inserite in Gradle si integrano, modificano ed eventualmente sostituiscono quelle nel Manifest.

```
buildTypes { NamedDomainObjectContainer<BuildType> it ->
    release {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
    }
}
```

Impostando a true l'opzione minifyEnabled rimuove risorse e codice del progetto che non sono utilizzate nell'APK finale per renderlo il più piccolo possibile.
ProGuard è un tool che permette ottimizzazioni ed offuscamento del codice della .apk

- a causa di queste ottimizzazioni il build è più lento, pertanto va evitato di utilizzarlo per il build di debug, ma solo per quello di release
- si può usare e/o disabilitare impostando useproguard false.

Per aggiungere un build type è sufficiente scrivere nel build.gradle del modulo

```
buildTypes {
    release {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
    }
    extra {
        minifyEnabled true
        useProguard false
        proguardFiles getDefaultProguardFile('proguard-android.txt')
    }
}
```

sono due tipi diversi di build, sono praticamente due app diverse tra loro. Il codice interno è uguale, ma viene impacchettata in maniera differente

quella di debug e release ci sono sempre. Quella di debug non viene distribuita

Es. build.gradle tramite la proprietà suffix modifica il package dell'app.

La configurazione e le risorse di un build type si impostano a livello di file e cartelle: si può creare in 'src' una cartella con struttura uguale a 'main', il nome della build e con solo classi java nuove. In riferimento all'esempio, andremo a creare una cartella di nome perf, con le sottocartelle java e res.
Il nuovo build type vedrà anche variabili e classi presenti in 'main' (**condivisione**), ma non le può sovrascrivere o modificare se già esistenti. Dunque, nella build type si mettono solo le cose nuove.
Mentre, il Manifest e le risorse di tipo valore vengono fuse (**merge**) con quelle presenti nel 'main', tutte eccetto quelle di tipo layout e le immagini (drawable). Ad esempio, le due versioni dell'app possono utilizzare loghi differenti.
Le **build flavor** permettono di creare varianti dell'app con configurazioni diverse per funzionalità, pubblico target o modalità di distribuzione. Sono app diverse che hanno in comune risorse e codice (es. icona diversa, grafica diversa, web server o DB alternativi).
Quando creiamo le build type, per il play store stiamo creando la stessa app. Cosa che non avviene, invece, con i build flavor.
Build type e build flavor permettono la creazione di versioni diverse di un'applicazione secondo due dimensioni ortogonali (**build variant** = build type[1..n] * build flavor[1..n]).
Es. si supponga di avere un'app con tre configurazioni, da distribuire in due versioni per i diversi clienti di **AziendaA** e **AziendaB**.
Si compila la versione **debug**, **performance** e **release** di entrambe tenendo logicamente separate le risorse per le due aziende, cioè:

- **buildTypes:** { debug{...}, release{...}, performance{...} }
- **productFlavors:** { AziendaA{...}, AziendaB{...} } l'applicationId sarà diverso.

Dunque, con build type si parla della struttura del progetto a livello di file e cartelle. Il defaultConfig non cambia. Mentre, con le build flavor si ha a che fare con il significato che si dà all'applicazione.

```
android {
    compileSdkVersion 34

    defaultConfig {
        minSdkVersion 21
        targetSdkVersion 34
        versionCode 1
        versionName "1.0"
    }

    buildTypes {
        debug {
            applicationIdSuffix ".debug"
            debuggable true
        }
        release {
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
        performance {
            applicationIdSuffix ".performance"
            debuggable false
        }
    }

    productFlavors {
        AziendaA {
            applicationId "com.example.aziendaA"
            resValue "string", "app_name", "App AziendaA"
        }
        AziendaB {
            applicationId "com.example.aziendaB"
            resValue "string", "app_name", "App AziendaB"
        }
    }
}

dependencies {
    implementation 'androidx.appcompat:appcompat:1.6.1'
    implementation 'com.google.android.material:material:1.11.0'
}
```

Questa configurazione permette di creare sei varianti dell'app combinando i tre buildTypes con i due productFlavors, generando così: **AziendaA**Debug, **AziendaA**Performance, **AziendaA**Release, **AziendaB**Debug, **AziendaB**Performance e **AziendaB**Release.

Altro esempio

Immaginiamo un'app con due versioni: una gratuita con annunci pubblicitari (**free**) e una a pagamento senza annunci (**paid**), entrambe con configurazioni per debugging (**debug**) e distribuzione (**release**).

```
android {
    compileSdk 34
    defaultConfig {
        applicationId "com.example.app"
        minSdk 21
        targetSdk 34
        versionCode 1
        versionName "1.0"
    }
    buildTypes {
        debug {
            applicationIdSuffix ".debug"
            debuggable true
        }
        release {
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }
    flavorDimensions "version"
    productFlavors {
        free {
            dimension "version"
            applicationId "com.example.app.free"
            versionNameSuffix "-free"
            buildConfigField "boolean", "SHOW_ADS", "true"
        }
        paid {
            dimension "version"
            applicationId "com.example.app.paid"
            versionNameSuffix "-paid"
            buildConfigField "boolean", "SHOW_ADS", "false"
        }
    }
}
```

FlavorDimensions è una proprietà utilizzata per definire categorie o dimensioni logiche di product flavors. È utile quando si vogliono creare più varianti di build basate su diverse caratteristiche, come il tipo di mercato, il target di utenti, o il tipo di distribuzione.
In conclusione:

- **Build Types:** gestiscono il modo in cui la tua app viene costruita (debug o release).
- **Build Flavors:** creano varianti della tua app per scopi o utenti diversi (free vs premium).

Puoi combinare i due: ad esempio, una build di tipo Debug Free per testare la versione gratuita o una Release Premium per distribuire la versione completa.

Cosa sono i qualificatori di risorse, che vantaggi portano e a cosa servono? Spiegare come Android Studio li gestisce e fornire almeno un esempio di utilizzo

Le risorse sono descritte da opportuni file contenuti in directory predefinite all'interno della cartella **/src/main/res** e vengono gestite dal **Resource Manager** con la **classe R**, la quale funziona sfruttando un **id** che può essere utilizzato per riferirsi a una determinata risorsa. Infatti, la caratteristica fondamentale di tutte le risorse è quella di poter essere refenziate attraverso un'apposita costante di una classe R generata automaticamente in fase di build oppure da un'opportuna sintassi da utilizzare all'interno di documenti XML di altre risorse.

Con Android, per avere (da codice Java) il riferimento ad un elemento della UI (definito nel layout XML) si può fare:

final Button button = findViewById(R.id.bottone_prova);

Esistono diversi tipi di risorse, ciascuna contenuta in una cartella

- **drawable:** elementi grafici, differenziati per densità e risoluzione
- **layout:** xml per organizzare gli elementi nelle activity
- **menu:** xml con info dei menu
- **mipmap:** icone dell'app (ottimizzate per ridimensionamenti)
- **values:** testo, stringhe, unità di misura.

I nomi delle cartelle in 'res', per Android, rappresentano una **annotazione delle risorse** (esplica già il tipo di risorsa contenuta). Dunque, tramite il nome, stiamo suggerendo ad Android che cosa sarà contenuto in quella cartella e in che formato.

Infatti, Android usa i **qualificatori**, (<*nome_risorsa*>-<*qualificatore*>), i quali vengono aggiunti ai nomi delle directory delle risorse per identificare varianti specifiche. Android utilizza queste directory per selezionare automaticamente la risorsa appropriata in base alla configurazione del dispositivo.

Es. drawable-mdpi, elementi grafici da caricare per un device con schermo a media densità.

Oppure, la selezione della lingua in base a quella del dispositivo, tramite differenti file string.xml.

Una cartella con nome senza qualificatore è quella di **default** ed è utile per prevenire errori ed avere compatibilità in avanti. Infatti, nel caso mancasse la risorsa nella cartella associata ai display ad alta risoluzione, ad esempio, un dispositivo classificato come tale non andrà a prendere la versione relativa alla densità media, ma cercherà nella cartella di default.

Quando il dispositivo ha la necessità di utilizzare una particolare risorsa, inizia la ricerca partendo dal primo qualificatore della lista (MCC e MNC) per poi proseguire con i successivi in ordine di priorità, fino ad arrivare all'API Level.

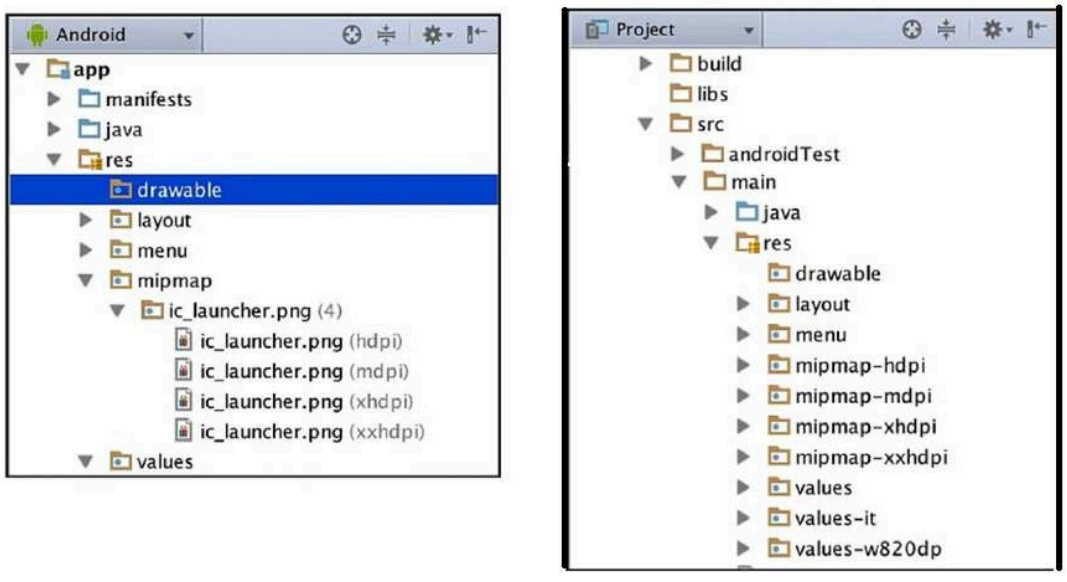


Figura 3.2 Le risorse relative alle immagini.

Come possiamo notare, sono state create quattro diverse cartelle che iniziano per mipmap e che sono seguite da un qualificatore. In questo caso si tratta di una stringa che identifica una particolare risoluzione del display che andrà a visualizzare la risorsa stessa. Vediamo poi come in ogni cartella vi sia un'immagine, relativa alla nostra icona, rappresentata da un file con lo stesso nome. È importante sottolineare, come la risorsa per la piattaforma sia una

sola, ovvero quella associata alla costante **R.mipmap.ic_laucher** che potremo utilizzare nel nostro codice Java, oppure associata alla stringa **@mipmap/ic_launcher** che potremo invece utilizzare all'interno di altri file che descrivono le risorse.
Sarà poi il particolare dispositivo che andrà a prendere la versione dell'immagine corrispondente alla risoluzione del proprio display.

CAPITOLO 5 - Activity

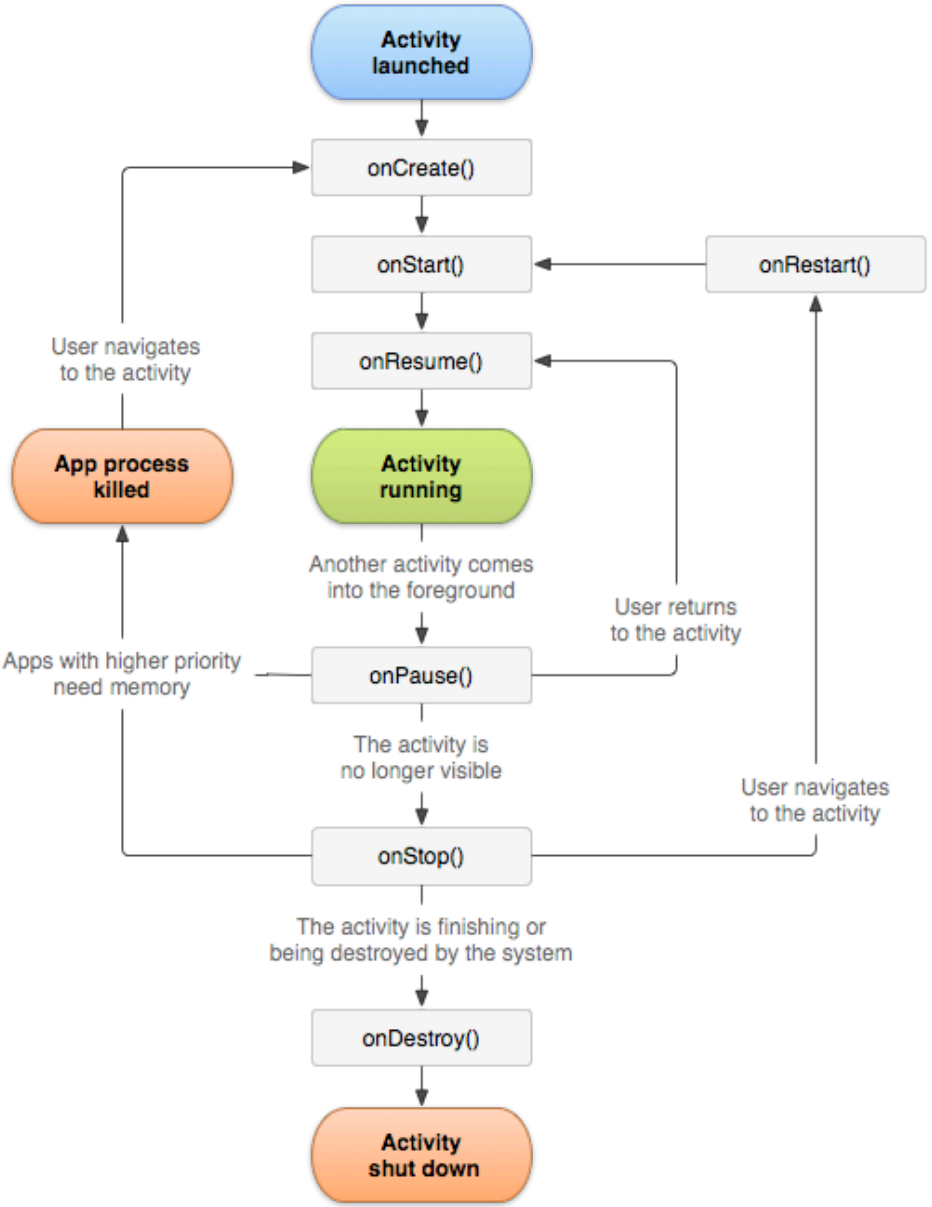
Cos'è un Activity?

In Android ciascuna schermata è rappresentata da una **Activity** che viene descritta da una specializzazione diretta o indiretta dell'omonima classe del package android.app.
Ogni volta che un'activity apre un'altra activity, l'activity chiamante è inserita nel **backstack** di Android (gestione **LIFO**) ed è messa in stop. Pertanto, tutte le attività di un particolare **task** vengono organizzate in una struttura a stack, con in testa l'activity con cui l'utente può interagire in quel momento, che dovrà necessariamente essere visibile. Durante l'interazione, lo stack modifica la sequenza, la visibilità ed il focus delle attività.
Ogni cambio di stato è notificato da un evento (**callback**), il quale si può intercettare per eseguire azioni compensative. I due metodi più importanti sono **onCreate()** e **onPause()**.
Inoltre, è possibile richiamare al servizio di una Activity un'app esterna installata nel device su cui sta girando l'app:

- al momento della chiamata, l'istanza dell'app viene **inglobata** nella nostra app e diventa un nuovo task
- Android permette la comunicazione in modo da poter inviare e ricevere dati dall'app esterna inglobata.

Illustrare il ciclo di vita di un'Activity, specificando i metodi di callBack, e in che modo si può forzare la chiusura dell'Activity.

Il diagramma seguente mostra i percorsi di stato importanti di un'Activity. I rettangoli rappresentano **metodi di callback** che puoi implementare per eseguire operazioni quando l'Activity si sposta tra stati. Gli ovali colorati sono gli **stati principali** in cui può trovarsi l'Activity.



In generale, lo svolgimento del ciclo di vita di un'activity si presenta in questo modo:

- **onCreate():** viene chiamato quando l'Activity viene creata per la prima volta ed è sempre seguito da **onStart()**. Può essere considerato l'equivalente del metodo main per una Activity, infatti, inizializza lo stato della classe, ovvero alloca memoria, imposta il layout e i riferimenti alla UI, gestisce il Bundle per eventuali stati precedenti (dopo un kill), ovvero se l'Activity è stata terminata in precedenza (ad es. per mancanza di memoria). In questo momento viene creata la classe R che permette di accedere alle risorse.
  onCreate è un **metodo critico** per il sistema, deve essere eseguito velocemente, pertanto dovrà contenere poco codice e non dovrà includere operazioni pesanti. Infatti, se entro 5 secondi l'app non risponde il sistema operativo mostra una finestra **"Application Not Responding (ANR)"** chiedendo all'utente se vuole forzare la chiusura dell'activity(kill).
- **onStart():** dopo che il layout è stato visualizzato e i listener sono stati impostati, l'Activity diventa **visibile**, ma non ancora pronta per interazioni dirette con l'utente.

Vengono implementate tutte quelle funzionalità legate esclusivamente alla visualizzazione, ma non all'interazione con gli utenti.

- **onResume():** activity è visibile ed attivata con focus, pronta per andare in Run. Vengono implementate tutte le funzioni che sono legate all'effettivo uso da parte dell'utente (accesso a risorse come videocamera, suoni o animazioni). A questo punto, l'activity è in cima allo stack, pronta a ricevere l'input dall'utente.
- **onPause():** è stata chiamata una nuova activity, l'attuale non sarà più interattiva (focus) ma ancora visibile (parzialmente, dialog). L'activity perde lo stato di primo piano, non è più focalizzabile. Utile per salvare uno stato persistente (qui sono richiamati i metodi del salvataggio stato)
- **onStop():** activity non è più attiva nè visibile, è stata messa in pila a livello inferiore. Ciò può accadere perché una nuova activity viene avviata in cima. Quando le activity sono terminate da Android potrebbe non essere chiamata (es. quando si ha bisogno di memoria), quindi è bene non inserire dati persistenti in questo metodo [vedi il doppio cammino process kill]
- **onRestart():** è alternativo al metodo onCreate(), per ripristinare dalla pila una activity già creata ma poi disabilitata.
- **onDestroy()**: l'activity non serve più, libera risorse del sistema.

Tra onStart() e onResume abbiamo **onRestoreInstanceState()**, mentre su onPause() abbiamo **onSaveInstanceState()**.
Se il sistema di gestione delle code di Android decide che è stata allocata troppa memoria, potrebbe chiudere l'activity.
L'**app process killed** è uno strumento usato da Android per liberarsi di quelle Activity che si si trovano nello stato di onPause o onStop. Questo avviene perché il sistema ha bisogno di memoria e quindi distrugge le Activity che in quel momento non stanno interagendo con l'utente.
Per forzare la chiusura di una Activity si utilizza il metodo **finish()**, il quale termina l'Activity corrente e la rimuove dallo stack, anche se in generale ci pensa Android a chiuderle quando è necessario.

Spiegare cosa succede all'app quando il dispositivo è ruotato e in che modo la UI è interessata

Quando il dispositivo viene ruotato, Android distrugge e ricrea l'Activity (e gli eventuali Fragment) per adattare l'app alla nuova configurazione, caricando le risorse appropriate. Questo processo, però, può causare la perdita del contenuto delle variabili e dello stato dell'Activity. Per evitare questo problema, Android fornisce un meccanismo per salvare e ripristinare lo stato dell'Activity tramite il metodo **onSaveInstanceState()**.
Dunque, quando il dispositivo viene ruotato:

1. Android salva lo stato dell'Activity nel metodo onSaveInstanceState().
2. L'Activity viene distrutta e passa attraverso onDestroy().
3. Viene creata una nuova istanza dell'Activity (con il nuovo orientamento) ed eseguito onCreate().
4. Durante onCreate(), lo stato precedentemente salvato viene ripristinato.
5. L'Activity passa attraverso onStart() e onResume(), diventando nuovamente interattiva.
6. L'utente può continuare a interagire con l'app senza perdere i dati precedenti.

Oltre a onCreate(), è possibile ripristinare lo stato anche nel metodo **onRestoreInstanceState()**, che viene chiamato subito dopo onStart().
Questo meccanismo garantisce una transizione fluida tra le configurazioni, senza che l'utente perda dati o debba reinserire informazioni.

Spiegare il ruolo del metodo di callback onSaveInstanceState()

In ambiente Android, un'operazione come la rotazione del dispositivo (da portrait a landscape) è considerata una variazione di configurazione, al pari di una modifica della lingua. Questo tipo di evento provoca il riavvio dell'Activity, **causando un reset dello stato**.
Ogni volta che ciò accade, è necessario salvare e ripristinare lo stato dell'Activity per garantire un'esperienza utente coerente.
Il sistema ci mette a disposizione il metodo **onSaveInstanceState(Bundle outState)**, che viene invocato automaticamente prima dell'eliminazione dell'Activity, sia a causa di cambiamenti di configurazione, sia per la richiesta di memoria da parte del SO, il quale può distruggere un activity.
La chiamata a onSaveInstanceState() non è garantita in tutte le situazioni, quindi non deve essere utilizzata per la persistenza a lungo termine (per questo scopo si usano SharedPreferences, file o database).
Es. Se Activity1 chiama Activity2, ma mentre si interagisce con Activity2 il sistema rimuove Activity1, quando l'utente preme su back Activity1 viene ricreata grazie ai dati salvati proprio nella chiamata a onSaveInstanceState(Bundle outState).
Per salvare lo stato si fa override del metodo onSaveInstanceState e al suo interno si implementa il salvataggio delle variabili di stato (es. valori per un calcolo mance) nel parametro Bundle, il quale memorizza tipi di dati primitivi come chiave-valore (KEY, VALUE).

```
@Override
protected void onSaveInstanceState(Bundle outState){
    super.onSaveInstanceState(outState);
    outState.putDouble(BILL_TOTAL, currentBillTotal);
    outState.putInt(CUSTOM_PERCENT, currentCustomPercent);
}
```

Android fornirà automaticamente outstate ad **onCreate(Bundle savedInstanceState**) se richiesto (in input). Il parametro Bundle **non è null** solo nel caso in cui ci si trovi in una fase di ripristino. Per recuperare i valori dal Bundle si scrive
this.mPerc = savedInstanceState.getInt(BILL_TOTAL)
dove mPerc è una variabile di tipo Int e BILL_TOTAL è la KEY con cui si è salvato il VALUE di un dato di tipo INT.

CAPITOLO 6 - I fragment

Che cosa è un Fragment e come lo si può implementare?

Il **Fragment** è un componente, dotato di ciclo di vita inserito e coordinato con quello della Activity ospitante, che permette una migliore scomposizione della UI di un'applicazione, in modo da renderla adatta all'esecuzione sia su smartphone che su tablet.

Si tratta di un componente diverso dall'Activity, che consente di gestire delle sotto-attività non solo per quello che riguarda la loro UI, ma anche la loro history, quel meccanismo che permette di mantenere lo stato di navigazione.
Come per le Activity, anche nei fragment ci sono alcuni metodi di callback i quali verranno invocati in corrispondenza di particolari stati del ciclo di vita dei Fragment.
Es. (Visualizzatore libri) Si consideri una progettazione basata su due Activity

- **Activity A:** mostra l'elenco dei titoli di opere e consente di visualizzarle in sequenza e di selezionarne una (dettagli a richiesta)
- **Activity B:** è invocata quando l'utente ha selezionato un titolo da A e mostra il dettaglio dell'opera scelta, con una citazione (permettendo ulteriori interazioni, seleziona, etc.).

Al posto di utilizzare due activity, è possibile utilizzare due Fragment contenuti in una Activity. Infatti, un **Activity può contenere n Fragment**.
Utilizzando un tablet (ma anche un device con orientamento landscape) si ha a disposizione più spazio sullo schermo, non è necessario passare continuamente tra Activity A e B (con l'uso del pulsante back o con Intent).
Si consideri quindi una diversa progettazione tramite doppio pannello (*approccio Master/Detail*)

- il primo è visibile sulla parte sinistra dello schermo e mostra l'elenco delle opere
- alla selezione di un titolo dall'elenco, il pannello elenco resta visibile, ma un secondo pannello si espande verso destra ed affianca il primo per fornire il dettaglio della citazione dell'opera selezionata
- non c'è più il passaggio tra Activity: i due pannelli interattivi o Fragment appartengono ad un unica Activity che li ospita e gestisce.

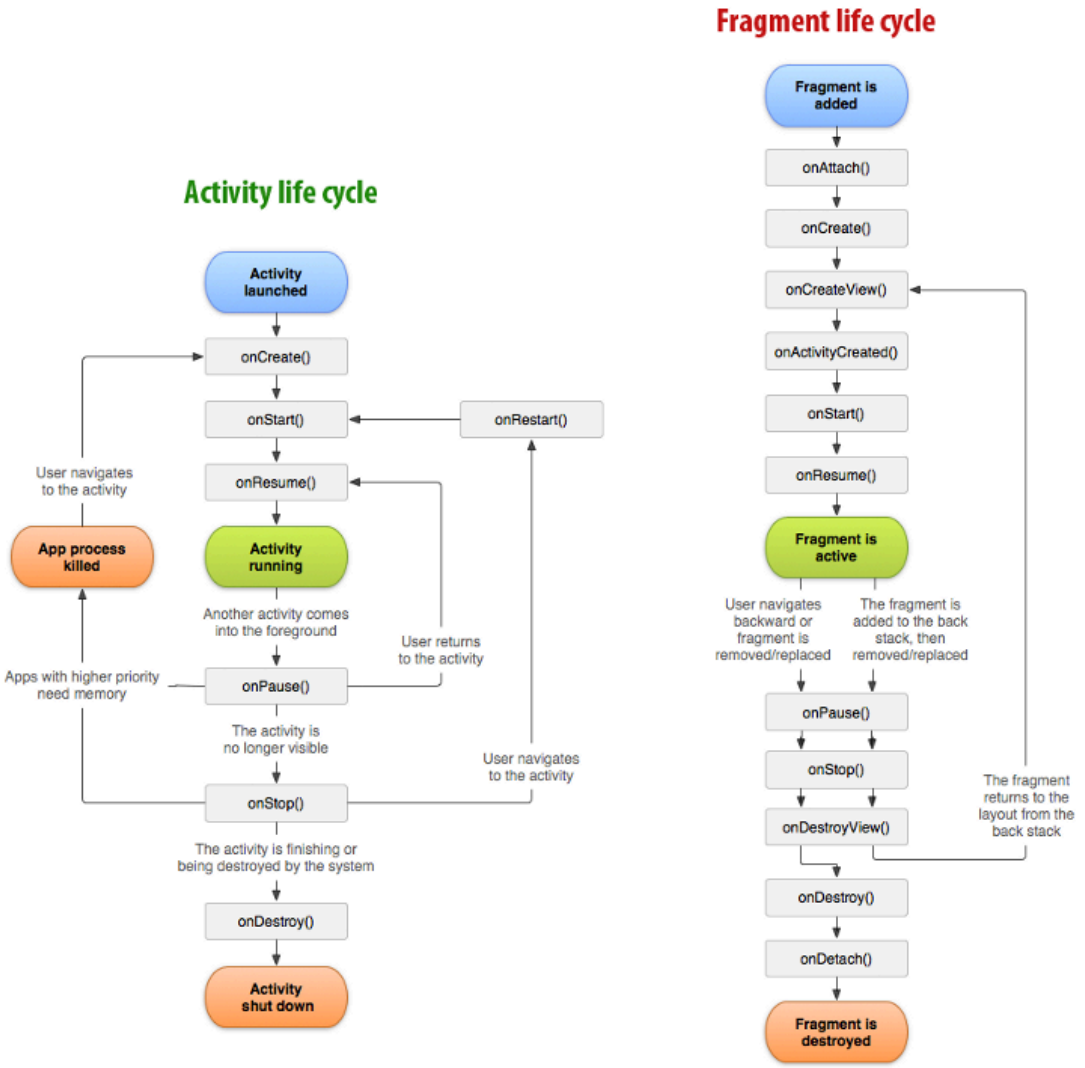
Vantaggio 1: si risparmia la creazione e distruzione di Activity che prima Android doveva gestire durante il task elenco-libro-selezione-mostra dettagli. Si riducono le in pila e fuori pila delle Activity e si ha una migliore gestione della memoria e scambio di dati.
Vantaggio 2: la user experience e le modalità di interazione sono più ricche e performanti.
Vantaggio 3: il riuso, in più Activity diverse o più volte nella stessa Activity.

Disegnare ed illustrare il ciclo di vita del Fragment ed i metodi di callback.

Il Fragment è creato in Android Studio con una struttura molto simile alla Activity

- ha un layout specificato con un file risorsa XML (in \res)
- ha un file java per il codice e gli stati.

Un Fragment ha il suo ciclo di vita e riceve chiamate ai vari metodi di callback degli stati, ma può esistere solo se ospitato (attach) in una Activity. Dunque, è strettamente correlato e coordinato con l'activity che lo ospita (**es.** se l'activity ospitante va in pausa, lo fa anche il fragment).



Essendo il Fragment contenuto in una Activity, i due cicli di vita sono certamente collegati fra loro. Il metodo di callback **onAttach()** associa il Fragment all'Activity contenitore e fornisce un suo riferimento.
Non appena un Fragment è aggiunto ad una Activity, Android invia il messaggio di callback **onCreateView()**, dove si fa l'inflate del layout. Poi chiama il metodo **onViewCreated(Bundle)** che contiene la logica che opera sugli elementi della View e gestisce un Bundle per ricreare lo stato degli elementi.
Quando viene richiesta la visualizzazione di un Fragment, esso segue un procedimento analogo a quello visto per le Activity, ovvero si ha l'invocazione del metodo **onStart()**, in corrispondenza della visualizzazione della UI associata e quindi del metodo **onResume()** quando tale UI diventa attiva e l'utente può interagirci.
Dopo un'azione dell'utente, un Fragment può essere rimosso dalla UI attualmente visualizzata per l'aggiunta di un altro con un dettaglio maggiore di informazioni. L'Activity contenitore rimarrà attiva, mentre il Fragment passerà prima nello stato di **PAUSED** (**onPause()**) e poi in quello di **STOPPED** (**onStop()**).
Anche la rimozione della View del Fragment è gestita in un callback apposito: **onDestroyView()**.
Infine, il metodo **onDetach()** stacca il Fragment dalla corrispondente Activity.

Se l'Activity in onStop() sta per essere eliminata, il Fragment riceve onDestroyView(), ma può essere recuperato insieme alla Activity dal sistema Android.
Il Fragment è strettamente correlato e coordinato con l'Activity che lo ospita.
Quando un'Activity è creata, il Fragment può ricevere diverse chiamate in base allo stato (coordinamento): onAttach() → onCreate() → onCreateView() → …
Gli **onSaveInstanceState(Bundle)** (che nel caso del ciclo di vita del Fragment si trova dopo onStop()) della Activity e del Fragment sono collegati tra loro, se viene chiamato dall'Activity, allora lancerà anche quello relativo al Fragment.
Se l'Activity entra in **onStart()**, diventando visibile, il fragment riceve la chiamata ad onStart(), ma del ciclo di vita del Fragment e non dell'Activity.
Se l'Activity riceve onResume, onPause e onStop anche il Fragment riceverà questi metodi, ma relativi al suo ciclo di vita.

Coordinamento tra Activity e Fragment

Il coordinamento tra un'Activity e un Fragment avviene attraverso il **FragmentManager**, che permette di:

- aggiungere un Fragment **staticamente** nel layout XML
- aggiungere un Fragment **dinamicamente** tramite codice, usando una **Transazione**.

In entrambi i casi, non appena il Fragment viene aggiunto, Android invia il messaggio di callback onCreateView().
Aggiunta statica:
Es. Il layout di una Activity di tipo LinearLayout può contenere due tag **<fragment>** dentro i quali vi è un attributo **class,** il quale fornisce il nome del file java che implementa il Fragment. All'interno del file del Fragment, onCreateView() fa Inflate() per caricare il layout del Fragment.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:baselineAligned="false"
    android:orientation="horizontal" >

    <fragment
        android:id="@+id/titles"
        android:layout_width="0px"
        android:layout_height="match_parent"
        android:layout_weight="1"
        class="course.examples.Fragments.StaticLayout.TitlesFragment" />

    <fragment
        android:id="@+id/details"
        android:layout_width="0px"
        android:layout_height="match_parent"
        android:layout_weight="2"
        class="course.examples.Fragments.StaticLayout.QuotesFragment" />

</LinearLayout>
```

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {

    return inflater.inflate(R.layout.quote_fragment, container, false);
}
```

```
public class QuoteViewerActivity extends Activity implements
        ListSelectionListener {

    public static String[] mTitleArray;
    public static String[] mQuoteArray;
    private QuotesFragment mDetailsFragment;

    private static final String TAG = "QuoteViewerActivity";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mTitleArray = getResources().getStringArray(R.array.Titles);
        mQuoteArray = getResources().getStringArray(R.array.Quotes);

        setContentView(R.layout.main);

        mDetailsFragment = (QuotesFragment) getFragmentManager()
                .findFragmentById(R.id.details);
    }
```

mDetailsFragment è un riferimento al Fragment istanziato nell'XML tramite la classe 'QuoteFragment'.

Aggiunta dinamica

Quando un'Activity è in esecuzione si può inserire un Fragment nel suo layout con una Transazione (begin - commit). Le transazioni garantiscono che le operazioni vengano fatte in modo atomico.

Per fare ciò, è necessario eseguire quattro operazioni:

1. si ottiene il riferimento al FragmentManager:
   FragmentManager fragmentManager = getFragmentManager();
2. si chiama il metodo beginTransaction nel fragmentManager che restituisce un elemento FragmentTransaction
   FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
3. si chiamano i metodi di aggiunta dei Fragment al fragmentTransaction a cui si passa l'ID del frameLayout e il Fragment che quella View conterrà
   fragmentTransaction.add(R.id.title_frame, new TitlesFragment());
   fragmentTransaction.add(R.id.quote_frame, mQuoteFragment);
4. infine si chiama il messaggio di esecuzione (metodo commit)
   fragmentTransaction.commit();
   quando tutte le operazioni sono state registrate all'interno della transazione, il metodo commit le rende effettive.

Nel setContentView il layout non ha più gli elementi Fragment ma due subView di tipo FrameLayout, il quale serve a riservare spazio nel layout che sarà occupato dal Fragment.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/activityFrame"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >

    <FrameLayout
        android:id="@+id/title_frame"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1" >
    </FrameLayout>

    <FrameLayout
        android:id="@+id/quote_frame"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="2" >
    </FrameLayout>

</LinearLayout>
```

Si passa un oggetto fragment istanziato o un new <nomeClasse>.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    mTitleArray = getResources().getStringArray(R.array.Titles);
    mQuoteArray = getResources().getStringArray(R.array.Quotes);

    setContentView(R.layout.main);

    FragmentManager fragmentManager = getFragmentManager();
    FragmentTransaction fragmentTransaction = fragmentManager
            .beginTransaction();
    fragmentTransaction.add(R.id.title_frame, new TitlesFragment());
    fragmentTransaction.add(R.id.quote_frame, mQuoteFragment);
    fragmentTransaction.commit();
}
```

Con i fragment gestiti tramite transazione:

- si rende l'interfaccia più fluida e si migliora l'uso dello spazio disponibile
- si creano UI dinamiche, ad es. nello stesso spazio del layout si possono inserire fragment diversi a seconda dei casi.

In che modo Activity e Fragment comunicano?

Da Activity a Fragment: una Activity può chiamare metodi dei fragment, acquisendo il riferimento al Fragment mediante il FragmentManager.
Si usano i metodi **findFragmentById()** o **findFragmentByTag()**.
Es. *ExampleFragment fragment = (ExampleFragment) getFragmentManager().findFragmentById(R.id.example_fragment);*
in questo modo può richiamare sull'oggetto ottenuto i metodi del Fragment.
Da Fragment ad Activity: il Fragment accede all'istanza dell'Activity con **getActivity()** e poi può effettuare vari task. Può ad esempio trovare una view nel layout dell'Activity che lo contiene: *View listView = getActivity().findViewById(R.id.list);*
In questo modo però si ottiene un riferimento che non permette di accedere ai metodi specifici di una nostra activity. Per farlo basta creare una variabile d'istanza del tipo specifico della nostra activity, che verrà inizializzata nel metodo onAttach(Context context).

CAPITOLO 7 - Intent

Si supponga di aver realizzato un'app con due Activity, come si stabilisce l'Activity di avvio? E come si passa all'altra Activity?

Per stabilire quale Activity deve essere avviata per prima in un'app Android, devi specificarlo nel file AndroidManifest.xml. Questo si fa dichiarando un'<activity> con un <intent-filter> contenente l'azione MAIN e la categoria LAUNCHER.

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

Al primo avvio di una Activity corrisponde una chiamata a onCreate(), il punto di partenza.
Per lanciare una specifica activity, se ne indica la classe: **intent esplicito**

```
Intent intent = new Intent(this, miaActivity.class);
startActivity(intent);
```

Per lanciare una activity generica, se ne indica l'azione: **intenti implicito**

```
Intent intent = new Intent(Intent.ACTION_...);
startActivity(intent);
```

Alcune Activity hanno senso solo se possono inviare un risultato di qualche tipo a chi le ha invocate. Es. l'Activity di sistema per scegliere un contatto dalla rubrica

```
Intent intent = new Intent(...);
startActivityForResult(intent, codice_richiesta);
```

Al termine dell'Activity lanciata, viene invocato il metodo onActivityResult() del chiamante, con argomenti che incapsulano la risposta.

Spiegare la differenza tra Intenti impliciti ed espliciti

Gli Intent si dividono in:

- **espliciti:** si specificano le componenti attraverso il nome della classe (presenti nell'app, ad esempio altre Activity) da far partire
- **impliciti:** non chiamano una componente specifica, ma dichiarano una azione generale da fare, così che possa essere gestita da un'altra componente (anche esterna, un servizio o altra app Android, etc.).

L'Intent implicito specifica un'azione che invoca un'app sul device che può effettuarla. L'istanza dell'app invocata è inglobata nell'app chiamante, come se ne fosse parte. Dunque, si utilizza quando altre app possono fare l'azione al posto dell'Activity chiamante. **Es.** se si

ha un contenuto da condividere, si crea un Intent con l'azione **ACTION_SEND** che specifica il contenuto da condividere.
I vantaggi sono:

- l'utente non deve imparare un'app nuova per fare un task
- il programmatore non deve reinventare la ruota
- app diverse di autori diversi comunicano tra loro (indirettamente) senza problemi di sicurezza e compatibilità tramite il sistema operativo.

Mentre, per comunicare quali Intent impliciti può **ricevere** l'app bisogna dichiarare uno o più **<intent-filter>** nelle Activity dichiarate nel Manifest.
Ciascun Intent Filter specifica il tipo di Intent che l'app può processare (a cui può rispondere) specificando i seguenti parametri: <action>, <data>, <category>.

- **<action>** è obbligatorio, dichiara l'azione accettata. Il valore è una stringa
  Es. <action android:name="android.intent.action.SEND"/>
- **<category>** dichiara la categoria dell'action fornendo ad Android informazioni su di essa
  <category android:name="android.intent.category.DEFAULT"/>
- **<data>** dichiara il tipo di dati accettato. Usa uno o più attributi URI e MIME per dire qual è la risorsa e la codifica
  <data android:mimeType="text/plain"/>

Dunque, è Android che si prende il compito di trovare l'Activity appropriata per gestire l'intent specifico e lo fa tramite il confronto con gli intent filter dichiarati negli AndroidManifest di tutte le app presenti nel sistema.

```xml
<activity android:name="MainActivity">
    <!-- This activity is the main entry, should appear in app launcher -->
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>

<activity android:name="ShareActivity">
    <!-- This activity handles "SEND" actions with text data -->
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="text/plain"/>
    </intent-filter>
    <!-- This activity also handles "SEND" and "SEND_MULTIPLE" with media data -->
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <action android:name="android.intent.action.SEND_MULTIPLE"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="application/vnd.google.panorama360+jpg"/>
        <data android:mimeType="image/*"/>
        <data android:mimeType="video/*"/>
    </intent-filter>
</activity>
```

Dire come funziona l'intent resolution e quando è applicato.

Supponiamo di realizzare una semplice applicazione che permetta l'inserimento di un URL e richiami un browser presente nel dispositivo per visualizzarlo. La nostra applicazione non dovrà sapere quale sarà il componente che andrà a visualizzare la pagina, ma creerà un **Intent**, all'interno del quale vi saranno le informazioni relative all'azione da eseguire e al tipo di dato su cui l'azione dovrà essere eseguita, che in questo caso sarà un semplice URL.
Una volta incapsulate le informazioni all'interno di un Intent, il sistema ci permetterà di lanciarlo nella speranza che venga raccolto da un qualche componente in grado di gestirlo.
A sua volta, il componente utilizzerà gli **Intent Filter** per dire al sistema di essere in grado di gestire alcune azioni su un particolare insieme di dati.
Al momento del lancio di un Intent il sistema eseguirà un'operazione di **Intent Resolution**, che consiste nel leggere tutti i manifest delle applicazioni installate sul device, valutare gli Intent Filter e scegliere quello o quelli più idonei. Il sistema esegue questa operazione in tre fasi:

1. **Action Test:** l'intent deve contenere un'azione compatibile con una di quelle dichiarate nell'Intent Filter dell'Activity. Se l'azione corrisponde a una delle azioni accettate dall'Intent Filter allora il test è superato. Se non c'è corrispondenza, l'Activity non sarà considerata.
2. **Category Test:** Ogni categoria specificata nell'Intent deve essere contenuta nell'IntentFilter. Se l'Intent non ha categorie, il test è superato automaticamente.
3. **Data Test:** Android verifica che il tipo di dato specificato nell'Intent corrisponda a quello supportato dall'Intent Filter.

Quando diverse app possono processare l'intent implicito l'utente può fare in modo di indicare l'app preferita, attraverso una finestra di dialogo che mostra le diverse app. Per mostrare tale finestra, si crea un altro Intent usando **createChooser()** e lo si passa a startActivity().

Spiegare cos'è un Intent e come viene costruito in Android

Un **Intent** è un oggetto che scambia messaggi tra componenti di app, utilizzato tipicamente in tre casi:

- **far partire un'Activity** (da una Activity A si fa partire una Activity B)
- **far partire un Service:** ovvero operazioni di background indipendenti da azioni dell'utente che partono in parallelo in modo asincrono, passando un Intent al metodo startService().
- **mandare un messaggio in Broadcast:** ovvero la notifica che nel sistema si è verificato un evento (batteria scarica, ricezione di una mail, etc.)

Per funzionare con una Activity si utilizzano i metodi:

- **startActivity():** fa partire una nuova Activity
- **startActivityForResult():** fa partire una Activity che restituisce un output
- **onActivityResult()**: ci verrà restituito un risultato.

App per la sveglia

Per creare una sveglia, si può usare **ACTION_SET_ALARM** (azione che permette di settare la sveglia), per specificare ora e messaggio da mostrare.

```
public void createAlarm(String message, int hour, int minutes) {
    Intent intent = new Intent(AlarmClock.ACTION_SET_ALARM)
            .putExtra(AlarmClock.EXTRA_MESSAGE, message)
            .putExtra(AlarmClock.EXTRA_HOUR, hour)
            .putExtra(AlarmClock.EXTRA_MINUTES, minutes);
    if (intent.resolveActivity(getPackageManager()) != null) {
        startActivity(intent);
    }
}
```

Nel manifest:

```
<uses-permission android:name="com.android.alarm.permission.SET_ALARM" />
<activity ...>
  <intent-filter>
    <action android:name="android.intent.action.SET_ALARM" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
```

modo per dire esplicitamente nel manifest che l'app chiederà ad android una funzione che richiede un permesso

Questo Intent richiede un permesso !

funzioni base di android

Coordinamento e comunicazioni tra le Activity. Dire in che modo si può passare un oggetto non elementare tra due Activity.

La comunicazione tra le Activity può avvenire in diversi modi:

1. Attraverso un **oggetto Bundle**, il quale viene usato come scatola di dati da passare tra le Activity
2. Attraverso un oggetto **Intent**, il quale fornisce i metodi **.putExtra(KEY, VALUE)** per poter passare in maniera semplice i dati ad una Activity che poi avvia programmaticamente. La nuova Activity può recuperare i dati creando un riferimento all'intent. **Es.** in onCreate() si crea un riferimento all'Intent (Intent src = getIntent();) che ha avviato quella stessa activity tramite **getIntent** e poi si usa getXXExtra(). Utile quando ci sono pochi dati da scambiare
3. Attraverso il passaggio di dati incapsulati in oggetti (**interfaccia classi Serializable**). Tuttavia, la serializzazione di Java dà problemi di performance in ambiente Android, pertanto è stata sostituita dalla **parcellizzazione**.
   La classe deve implementare l'interfaccia **android.os.Parcelable** e i metodi describeContents() e writeToParcel(Parcel, int). L'oggetto Parcel è dove scrivere i dati.

```
public void writeToParcel(Parcel dest, int flags) {
        dest.writeString(name);
        dest.writeFloat(latitude);
    ..... }
```

Viene invocata nel momento in cui si deve trasferire un oggetto. Il parametro Parcel rappresenta il contenitore dal quale andare a scrivere i vari parametri attraverso una serie di metodi del tipo writeXXX() che non associano il dato ad una chiave

Recuperare i dati: la lettura dei valori va fatta seguendo l'ORDINE di inserimento !

```
String nam = dest.readString();
String lat = dest.readFloat();
```

In che modo si recupera un valore di ritorno dalla chiamata di un Intent?

Nella comunicazione tra Activity A (chiamante) e Activity B (chiamata), è responsabilità dell'Activity B restituire un risultato all'Activity A.
Per farlo, l'Activity B utilizza il metodo **setResult()**, che imposta:

- Un **resultCode** (ad esempio, RESULT_OK per un risultato valido o RESULT_CANCELED per un'operazione annullata).
- Un **resultData**, ovvero un Intent che contiene i dati da restituire.

Dopo aver impostato il risultato, l'Activity B si chiude con **finish()**.

```
Intent resultIntent = new Intent();
// TODO Add extras or a data URI to this intent as appropriate.
resultIntent.putExtra("some_key", "String data");
setResult(Activity.RESULT_OK, resultIntent);
finish();
```

L'Activity A utilizza **startActivityForResult()** (metodo obsoleto) o **ActivityResultLauncher** (metodo moderno) per avviare l'Activity B e attendere un risultato.
Quando l'Activity B si chiude, l'Activity A riceve il risultato nel metodo **onActivityResult()** e lo elabora.

CAPITOLO 8 - I Permessi

Qual è la differenza tra i permessi normal e quelli dangerous? Quale componente effettua la verifica e quando?

Il permesso, in Android, viene usato per proteggere risorse e dati. Infatti, limita l'accesso a informazioni sull'utente, API sensibili ad eventuali addebiti e risorse costose (es. fotocamera).
Un'app dichiara i permessi di cui necessita nel Manifest, attraverso il tag **<uses-permission>**

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.example.snazzyapp">

    <uses-permission android:name="android.permission.SEND_SMS"/>

    <application ...>
        ...
    </application>
</manifest>
```

In questo caso l'app richiede i permessi per inviare SMS.
I permessi non rischiosi sono concessi dal sistema (di solito quelli del livello 0 - normale), in caso contrario devono essere concessi dall'utente, attraverso la gestione della richiesta.
Esistono due tipologie di permessi:

- Se la tua app elenca **permessi normal** nel file manifest (cioè permessi che non rappresentano un rischio significativo per la privacy dell'utente o il funzionamento del dispositivo), il sistema li concede automaticamente.
- Se, invece, la tua app elenca **permessi dangerous** nel file manifest (cioè permessi che potrebbero potenzialmente influire sulla privacy dell'utente o sul normale funzionamento del dispositivo), il sistema chiederà esplicitamente all'utente di concederli. Il modo in cui Android gestisce queste richieste dipende dalla versione del sistema operativo e dalla versione target del sistema specificata dalla tua app.

In Android 6.0+ (Marshmallow) i permessi dangerous sono richiesti a run-time e l'utente in qualsiasi momento può revocare tali permessi, pertanto l'app deve verificare i permessi ogni qualvolta debba utilizzare le risorse. Mentre, nelle versioni precedenti, i permessi erano verificati solo all'installazione, quindi bastava indicarli nel manifest.

Permessi normali

Non sono dei veri e propri permessi, ma delle funzionalità.
Nel manifest si dichiarano usando il tag <uses-feature>

```
<uses-feature android:name="android.hardware.camera" android:required="false" />
```

con l'attributo **Android:required="false",** consente l'installazione dell'app nel device anche se quel requisito non è presente (rischiando di influenzare negativamente la User Experience dell'utente). Se impostata a **true**, il Play Store non mostra l'app al dispositivo che non rispetta il requisito.

Permessi speciali e pericolosi

Per utilizzare i permessi "dangerous", è necessario dichiararli nel Manifest (**tag uses-permission**) e inviare un'Intent per richiedere l'autorizzazione all'utente. I permessi di tipo dangerous devono essere gestiti a runtime dallo sviluppatore, poiché l'utente ha la possibilità di revocarli in qualsiasi momento.
Per verificare se un permesso è stato concesso, si utilizza il metodo **ContextCompat.checkSelfPermission().**

```
// Assume thisActivity is the current activity
int permissionCheck = ContextCompat.checkSelfPermission(thisActivity,
        Manifest.permission.WRITE_CALENDAR);
```

Il PackageManager restituirà uno dei seguenti valori:

- **PERMISSION_GRANTED**: Se il permesso è stato concesso, si può proseguire con l'operazione richiesta.
- **PERMISSION_DENIED**: Se il permesso non è stato concesso, è necessario richiederlo esplicitamente all'utente.

È buona pratica spiegare all'utente il motivo per cui si sta chiedendo un permesso. Questo può essere fatto tramite il metodo shouldShowRequestPermissionRationale(), che mostra un pannello informativo. Inoltre, è importante informare l'utente sulle possibili conseguenze se il permesso non viene concesso, e fornire modalità alternative o soluzioni compensative per garantire il corretto funzionamento dell'app.

```
// Here, thisActivity is the current activity
if (ContextCompat.checkSelfPermission(thisActivity,
                Manifest.permission.READ_CONTACTS)
        != PackageManager.PERMISSION_GRANTED) {

    // Should we show an explanation?
    if (ActivityCompat.shouldShowRequestPermissionRationale(thisActivity,
            Manifest.permission.READ_CONTACTS)) {

        // Show an explanation to the user *asynchronously* -- don't block
        // this thread waiting for the user's response! After the user
        // sees the explanation, try again to request the permission.

    } else {

        // No explanation needed, we can request the permission.

        ActivityCompat.requestPermissions(thisActivity,
                new String[]{Manifest.permission.READ_CONTACTS},
                MY_PERMISSIONS_REQUEST_READ_CONTACTS);

        // MY_PERMISSIONS_REQUEST_READ_CONTACTS is an
        // app-defined int constant. The callback method gets the
        // result of the request.
    }
}
```

CAPITOLO 9 - L'interfaccia

Fornire una definizione di Layout e spiegare come vengono creati e gestiti in Android Studio. Fornire un esempio di un tipo di Layout noto aiutandosi con schemi e frammenti di codice

I **Layout** sono risorse che permettono di definire in modo dichiarativo l'interfaccia grafica di un'applicazione Android. Questi vengono specificati tramite file XML, utilizzando tag che descrivono l'aspetto e la disposizione degli elementi grafici. I file di Layout si trovano nella cartella res/layout/*.xml, ed è possibile farne riferimento

- in Java tramite id: R.id.<nomeLayout>
- in altre risorse: un_layout="@layout/<nomeLayout>"

Android supporta più versioni dello stesso layout per adattarsi a diverse condizioni, come: dimensioni dello schermo, tema, eventi specifici. L'adattamento avviene automaticamente grazie ai qualificatori di risorse, che permettono al sistema di scegliere il layout più adatto.
Tutti gli elementi dell'interfaccia di Android sono creati utilizzando oggetti View e ViewGroup

- **View:** rappresentano gli elementi visibili con cui l'utente può interagire. Occupano uno spazio rettangolare sullo schermo e sono responsabili per il loro stesso rendering e per la gestione degli eventi
- **ViewGroup:** oggetto che contiene altri oggetti View e ViewGroup per definire il layout.

Tra le View predefinite, ci sono:

- **Button:** View su cui l'utente può effettuare un click per eseguire azioni. Solitamente associato ad un listener che si mette in ascolto delle pressioni di tale Button.
- **ToggleButton:** è un tipo di Button che quando è premuto, resta premuto. Ha tipicamente due stati, checked e unchecked.
- **Checkbox:** casella selezionabile per scelte multiple
- **Rating Bar:** permette all'utente di assegnare un punteggio
- **AutoCompleteTextView:** mostra del testo editabile e la View dà suggerimenti all'utente.

Come avviene il rendering della UI? In particolare, com'è organizzato in Android lo spazio dedicato alle Views?

Le View sono organizzate tipicamente come alberi. Quando Android disegna le View sullo schermo scorre l'albero molte volte, prende le dimensioni delle View, le posizioni e poi le disegna.
Concettualmente esegue i seguenti passi:

- **measure** → prende le dimensioni delle View
- **layout** → una volta determinate le dimensioni, posiziona le View
- **draw** → disegna le View.

In genere non ci si preoccupa dell'ordine con cui Android scandisce l'albero, ma se si vogliono realizzare sottoclassi di View custom bisogna fare overriding di diversi metodi relativi alle View.
Per esempio:

- **onMeasure()**: determina le dimensioni di una View e dei suoi figli

- **onLayout():** è chiamato durante la fase di layout, assegna dimensione e posizione ad ogni figlio
- **onDraw():** chiamato nella terza fase della View, effettua il rendering dei suoi contenuti.

Ogni View in Android non è altro che uno spazio rettangolare che può essere specializzato in widget complessi. I vari widget sono contenuti in un layout genitore, che a seconda delle caratteristiche, dispone i propri figli in **modi ben definiti** (LinearLayout, TableLayout, Grid, ecc) o sulla **base di vincoli** (ConstraintLayout, RelativeLayout). Tutti i layout in Android sono **sottoclassi di ViewGroup** (a sua volta sottoclasse di View), il cui compito è quello di contenere altre view e specificare vincoli e parametri (LayoutParams).

Constraint Layout

Il **ConstraintLayout** è un avanzato ViewGroup introdotto in Android per creare interfacce **flessibili** ed **efficienti**, riducendo la nidificazione eccessiva di altri layout. A differenza di LinearLayout e RelativeLayout, permette di posizionare e ridimensionare i componenti attraverso vincoli (**constraints**), semplificando la gerarchia delle View e migliorando le prestazioni della UI.
Ogni elemento all'interno di un ConstraintLayout può essere vincolato ai bordi di altri elementi o direttamente al contenitore principale (**parent**). Ogni elemento deve avere almeno un vincolo orizzontale e uno verticale per evitare errori. Esistono diversi tipi di vincoli, ad es. Top_toBottomOf, Start_toStartOf, ecc. Questo consente di creare layout **dinamici** e **adattabili** senza la necessità di strutture annidate, migliorando la leggibilità e la gestione del codice.
Le sue caratteristiche principali sono:

- Posizionamento relativo: gli elementi possono essere vincolati ad altri elementi presenti nel layout
- Centratura: gli elementi possono essere centrati sia orizzontalmente che verticalmente (usando bias)
- Posizionamento circolare
- Margini: si può specificare la distanza tra i widget
- Gestione della visibilità: visible, invisible e gone

Linear Layout

Il **LinearLayout** è un tipo di ViewGroup che organizza i suoi elementi figli lungo un'unica direzione, che può essere orizzontale o verticale. L'orientamento viene definito attraverso l'attributo **android:orientation** nel file XML o tramite il metodo **setOrientation(int)** nel codice. Sono disponibili due possibili valori: horizontal, per disporre gli elementi in fila, e vertical, per allinearli in colonna.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent
   android:layout_height="match_parent"
   android:orientation="vertical" >
   <Button
      android:id="@+id/button1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/button_label_1" >
   </Button>
   <Button
      android:id="@+id/button2"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/button_label_2" >
   </Button>
</LinearLayout>
```

VARIANTE: **android:orientation="horizontal"**

Relative Layout

Il **RelativeLayout** è un tipo di ViewGroup che permette di posizionare ogni componente in relazione agli altri elementi presenti nel layout o rispetto al contenitore principale. Le View figlie vengono disposte sulla base dei vincoli definiti tra di loro e rispetto alla View genitore. Android segue l'ordine specificato nel file XML, ma il calcolo delle posizioni avviene solo dopo aver analizzato l'intera struttura del layout. Questo meccanismo consente di fare riferimento a elementi ancora non definiti nel codice XML.
Per posizionare un elemento in base a un altro, si utilizzano attributi specifici. Ad esempio, **android:layout_toRightOf="@id/usernameLabel"** permette di posizionare una View alla destra di un'altra identificata dall'ID usernameLabel. In modo simile, esistono altri attributi come layout_below, layout_alignParentStart e layout_centerInParent, che consentono di definire con precisione la disposizione degli elementi all'interno del RelativeLayout.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:id="@+id/relativeLayout"
   android:layout_width="match_parent"
   android:layout_height="match_parent" >
   <EditText
      android:id="@+id/username"
      android:layout_width="wrap_content"   android:layout_height="wrap_content"
      android:layout_alignParentRight="true"
      android:layout_toRightOf="@+id/usernameLabel"
      android:hint="@string/username_hint" >
   </EditText>
   <TextView
      android:id="@+id/usernameLabel"
      android:layout_width="wrap_content"   android:layout_height="wrap_content"
      android:layout_alignBaseline="@+id/username"
      android:text="@string/username_label" >
   </TextView>
   <EditText
      android:id="@+id/password"
      android:layout_width="wrap_content"   android:layout_height="wrap_content"
      android:layout_alignLeft="@+id/username"
      android:layout_alignParentRight="true"
      android:layout_below="@+id/username"
      android:hint="@string/password_hint" >
   </EditText>
   <TextView
      android:id="@+id/passwordLabel"
      android:layout_width="wrap_content"   android:layout_height="wrap_content"
      android:layout_alignBaseline="@+id/password"
      android:text="@string/password_label" >
   </TextView>
</RelativeLayout>
```

Composizione della UI

L'interfaccia utente (UI) di un'app Android è strutturata come un **albero gerarchico**, in cui

- i **nodi intermedi** sono oggetti della classe ViewGroup, che fungono da contenitori
- le **foglie** sono oggetti della classe View, che rappresentano i singoli elementi visibili e interattivi.

Ogni View in Android è una classe Java il cui nome corrisponde al relativo tag XML utilizzato nella definizione del layout.
L'organizzazione visiva degli elementi viene gestita tramite un Layout, che determina la disposizione e il comportamento delle View all'interno del loro contenitore (ViewGroup).
Un ViewGroup può contenere un numero qualsiasi di View e altri ViewGroup, permettendo la creazione di UI complesse.
I ViewGroup predefiniti sono:

- **RadioGroup:** contiene un insieme di checkBox o radioButton di cui solo uno può essere selezionato. Es. Un'app che chiede l'età e dà la possibilità di scelta tra diversi intervalli, solo una checkbox deve essere selezionata.
- **TimePicker:** Il TimePicker è una classe che estende ViewGroup che permette di selezionare l'ora del giorno in modo vincolato senza farla inserire manualmente dall'utente (condizionando l'input ed evitando eventuali errori). Molto spesso è

raffigurato come un orologio sul quale si seleziona prima l'ora e poi i minuti oppure con due cilindretti per ora e minuti.
Si consiglia di utilizzare un DialogFragment per ospitare un TimePicker.

```
public static class TimePickerFragment extends DialogFragment
                    implements TimePickerDialog.OnTimeSetListener {

    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        // Use the current time as the default values for the picker
        final Calendar c = Calendar.getInstance();
        int hour = c.get(Calendar.HOUR_OF_DAY);
        int minute = c.get(Calendar.MINUTE);

        // Create a new instance of TimePickerDialog and return it
        return new TimePickerDialog(getActivity(), this, hour, minute,
                DateFormat.is24HourFormat(getActivity()));
    }

    public void onTimeSet(TimePicker view, int hourOfDay, int minute) {
        // Do something with the time chosen by the user
    }
}
```

- **DatePicker:** Il DatePicker è una classe che estende ViewGroup che permette di selezionare una data in modo **vincolato** senza farla inserire manualmente dall'utente (condizionando l'input ed evitando eventuali errori). Molto spesso è raffigurato come un calendario sul quale si seleziona la data oppure con tre cilindretti per giorno mese anno.
  Si consiglia di utilizzare un DialogFragment per ospitare un DatePicker.

```
public static class DatePickerFragment extends DialogFragment
                    implements DatePickerDialog.OnDateSetListener {

    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        // Use the current date as the default date in the picker
        final Calendar c = Calendar.getInstance();
        int year = c.get(Calendar.YEAR);
        int month = c.get(Calendar.MONTH);
        int day = c.get(Calendar.DAY_OF_MONTH);

        // Create a new instance of DatePickerDialog and return it
        return new DatePickerDialog(getActivity(), this, year, month, day);
    }

    public void onDateSet(DatePicker view, int year, int month, int day) {
        // Do something with the date chosen by the user
    }
}
```

- **WebView:** mostra pagine web.
- **MapView:** un ViewGroup che mostra mappe e consente di interagire con essa.
- **Gallery:** mostra un gruppo di dati in una lista scollabile orizzontalmente.

- **Spinner:** lista di elementi scrollabile all'interno di un AdapterView. L'utente clicca sul componente e dalla lista che compare seleziona l'elemento desiderato. I dati vengono gestiti da uno SpinnerAdapter.

Quali sono gli stati della proprietà "visibility" e a quale categoria di classi si riferisce?

In Android, la proprietà **visibility** determina se una View è visibile o meno all'interno dell'interfaccia utente. Questo attributo si applica a tutte le classi che derivano da View, come Button, TextView, ImageView e altri widget UI.
Una View può trovarsi in uno dei tre stati seguenti:

- **VISIBLE:** la view è completamente visibile e interattiva. Occupa spazio all'interno del layout (android:visibility="visible")
- **INVISIBLE:** la view è ancora presente nel layout, occupa spazio, ma non è visibile all'utente. Gli altri elementi mantengono la loro posizione come se la View fosse visibile
- **GONE:** la view non è visibile e non occupa più spazio nel layout. Gli elementi circostanti si ridistribuiscono per occupare il suo spazio.

Quando si specificano le proprietà layout_gravity e gravity e a cosa servono?

La proprietà **gravity** determina l'allineamento del contenuto all'interno di una View (es. textView), definendo la posizione degli elementi interni. Supporta valori come top, bottom, left, right, center, center_vertical e center_horizontal. Al contrario, **layout_gravity** specifica dove la View verrà posizionata all'interno del suo contenitore, simulando il comportamento di un oggetto che "cade" in base alla gravità. Le possibilità sono: top, center, bottom e fill (riempimento). Layout_gravity può essere utilizzata solo se il layout padre è un LinearLayout.

Quali meccanismi mette a disposizione Android per "parlare la lingua dell'utente" (es. francese per i francesi, italiano per gli italiani)?

Per la lingua, si sfrutta la capacità di Android di usare i qualificatori per caricare automaticamente le risorse adatte, riconoscendo le caratteristiche e quindi la lingua impostata sul device.
I file di stringhe creati hanno tutti lo stesso nome (strings.xml) e la stessa struttura, cambia solo la sotto cartella in /res che ha il qualificatore corrispondente alla lingua

- /values → è il default, ad es. inglese
- /values-es → lingua spagnola
- /values-it → lingua italiana.

/values/strings.xml : (default locale)

```
<resources>
    <string name="hello_world">Hello World!</string>
</resources>
```

/values-es/strings.xml :

```
<resources>
    <string name="hello_world">¡Hola Mundo!</string>
</resources>
```

Il contenuto dei tag varia in base alla lingua.

Plurals: dire come sono gestite le quantità nelle risorse e nel codice java.

I **Plurals** in Android servono per gestire automaticamente le **variazioni grammaticali** delle quantità in base alla lingua impostata sul dispositivo. Questo permette di ottenere la corretta forma singolare o plurale senza dover gestire manualmente le traduzioni, sfruttando i qualificatori di risorse.
Nel file XML delle risorse (res/values), i plurals vengono definiti come un insieme di stringhe associate a diverse quantità. Android supporta sei categorie grammaticali: zero, one, two, few, many e other. Tuttavia, non tutte le lingue usano tutte queste categorie; ad esempio, in italiano si distingue solo tra singolare (one) e plurale (other), mentre in altre lingue le regole possono essere più complesse.
Per recuperare dinamicamente la forma corretta di una stringa in base a un numero, si utilizza il metodo **getQuantityString()**.
Esempio in inglese

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <plurals name="numberOfSongsAvailable">
        <!-- As a developer, you should always supply "one" and “other” strings. Your translators will know which strings are actually needed for their language. Always include %d in "one" because translators will need to use %d for languages where “one” doesn't mean 1  -->
        <item quantity="one">%d song found.</item>
        <item quantity="other">%d songs found.</item>
    </plurals>
</resources>
```

Ad esempio, chiamando **getQuantityString(R.plurals.numberOfSongsAvailable, 1, 1)**, Android riconosce che per la lingua inglese l'input 1 corrisponde alla forma one e restituisce "1 song found" (il valore per il segnaposto %d proviene dal secondo numero di input). Se invece il numero fosse 2, il sistema restituirebbe "2 songs found".
Questa gestione automatica è particolarmente utile nelle applicazioni multilingua, evitando errori grammaticali e migliorando l’esperienza utente.

CAPITOLO 10 - Menu e barre

Differenza tra ActionBar e ToolBar

Entrambe le barre servono a gestire menu, comandi e funzionalità di navigazione, offrendo un'esperienza coerente all'utente.
L'**ActionBar** è la barra superiore di navigazione e controllo predefinita nelle app Android (API 11+). È strettamente legata all'Activity e fornisce:

- Un'icona o il logo dell'app.
- Il titolo della schermata.
- Un menu con opzioni aggiuntive.
- Supporto per la navigazione tramite tab o liste a tendina.

L'ActionBar aiuta l'utente a orientarsi nell'app e accedere rapidamente alle azioni principali. Tuttavia, non è personalizzabile come la Toolbar e richiede almeno API 21 per supportare il Material Design.
La **Toolbar** (androidx.appcompat.widget.Toolbar) è un widget indipendente, posizionabile ovunque nel layout e compatibile con API 7+. Può sostituire l'ActionBar tramite **setSupportActionBar(toolbar)**, offrendo maggiore flessibilità:

- Pulsante di navigazione personalizzato (es. tasto "Indietro").
- Icone, titolo e sottotitolo modificabili.
- Possibilità di includere altre View (es. campi di ricerca, pulsanti).
- Supporto per Material Design anche su versioni precedenti di Android.
- Utilizzabile anche in un Fragment, a differenza dell'ActionBar.

Quale criterio si suggerisce di usare per determinare la priorità degli elementi dell'action bar?

I pulsanti d'azione servono per le funzionalità più importanti, spesso quelle più utilizzate

- in base allo spazio disponibile il sistema mostrerà solo le azioni più importanti e le restanti saranno nascoste nell'**action overflow** (tre puntini sulla destra)
- l'action bar dovrebbe mostrare solo le azioni disponibili all'utente, se una funzionalità non è disponibile bisogna toglierla dal contesto, nascondendola, mai mostrarla disabilitata.

Lo **schema FIT** viene utilizzato per selezionare i pulsanti d'azione da inserire nell'ActionBar:

- **Frequenza:** quante volte viene utilizzata quella azione
- **Importanza:** rilevanza all'interno dell'applicazione
- **Tipicità:** se viene utilizzata anche in app simili.

Criterio di scelta: se almeno uno tra F, I e T è valido allora la funzionalità è indicata per essere un action button

Le tipologie di menu

Le Activity possono essere supportate dai **menu**, i quali possono essere molto diversi. E' possibile aggiungere elementi al menu e gestire i click sugli elementi del menu.
Alcuni tipi di menu sono:

- **Option:** mostrati quando l'utente preme il pulsante del menu. Dalla versione 3.0 è accessibile dalla barra dell'app. Per default gli elementi sono tutti nell'action overflow (i tre puntini).
- **Contestuali:** visibili solo quando l'utente tiene il touch su una View e sono contestuali a tale View.
    - Il **Floating Context Menu** compare al centro del display, ad esempio se si seleziona un elemento della UI. Offre azioni che influiscono su un elemento specifico nell'interfaccia utente. Viene spesso utilizzato con i fragment.
    - **Contextual Action Mode** è una barra personalizzata con operazioni da eseguire su uno o più elementi selezionati.
- **Sottomenu**: attivati quando l'utente tocca un elemento di menu visibile
- **Menu popup:** consistono in liste verticali di item ancorati alla View e invocati dal menu.

I menu si definiscono nelle risorse in **res/menu/nomefile.xml.**

CAPITOLO 11 - Material Design

Spiegare perché una delle linee guida del material design vieta torsioni e piegamenti degli elementi dell'interfaccia utente.

Una delle linee guida del Material Design **vieta torsioni e piegamenti** degli elementi dell'interfaccia utente perché il Material Design **si basa sulla metafora dei materiali**, che simula il comportamento della carta e penna nel mondo reale.
I materiali nel Material Design hanno proprietà fisiche specifiche:

- **Immutabilità e comportamento intrinseco:** gli elementi possono variare in larghezza e altezza, ma mantengono sempre lo stesso spessore (1dp).
- **Assenza di torsioni e piegamenti:** gli elementi possono cambiare forma in modi coerenti con le proprietà fisiche dei materiali, come unirsi, dividersi o auto-ripararsi, ma non possono torcersi o piegarsi, perché ciò violerebbe la coerenza visiva e il principio di realismo del design.
- **Movimento naturale:** le animazioni e le trasformazioni devono rispettare le leggi della fisica, per garantire un'esperienza d'uso intuitiva e coerente.

Questa scelta progettuale aiuta a mantenere un'interfaccia chiara, prevedibile e in linea con le aspettative dell'utente, evitando effetti innaturali o confusionari.

Spiegare il concetto di elevazione fornendo alcuni esempi. Dire anche se è rilevante il dispositivo su cui l'elevazione è utilizzata.

Il concetto di **elevazione** nel Material Design si riferisce alla distanza tra una superficie e un'altra lungo l'asse Z, che è ortogonale allo schermo e rivolto verso l'utente.
Principi chiave dell'elevazione:

- Tutti gli oggetti hanno uno spessore fisso di 1dp, ma possono essere posizionati a diverse elevazioni lungo l'asse Z.
- L'ombra è l'unico indicatore dell'elevazione, quindi la percezione della profondità si basa sulle variazioni di luce e ombra.

- **L’elevazione può essere statica o dinamica:** alcuni elementi mantengono sempre la stessa elevazione, mentre altri possono cambiare temporaneamente (es. un pulsante che si solleva quando viene premuto).

Esempio. FAB - Floating Action Button: ha una elevazione maggiore (8 dp) rispetto agli altri elementi dell’interfaccia per indicare che è un’azione principale.

Spiegare cosa si intende per ‘continuità visuale’ nel material design.

La **continuità visuale** nel Material Design si riferisce alla capacità dell’interfaccia di mantenere un **flusso coerente** e **comprensibile** durante le transizioni e le interazioni dell’utente. L’obiettivo è mantenere una connessione visiva tra gli elementi quando cambiano stato, dimensione o posizione, in modo che l’utente percepisca una trasformazione naturale e prevedibile, piuttosto che un cambiamento improvviso.
Gli oggetti devono muoversi in modo fluido e rispettare le leggi della fisica, evitando animazioni casuali o disorientanti.

Material Design e animazioni: cosa dicono le linee guida rispetto alle animazioni e all’organizzazione degli elementi della UI?

Le linee guida del Material Design stabiliscono che le animazioni devono essere **naturali**, **coerenti** e **significative**, contribuendo a migliorare l’esperienza utente senza distrarre o confondere.
Principi chiave delle animazioni:

- **L’utente inizia il movimento** → Le animazioni devono sempre rispondere alle azioni dell’utente, evitando movimenti casuali o inutili.
- **Movimenti realistici** → Le transizioni devono rispettare le leggi della fisica, considerando massa, accelerazione e inerzia. I movimenti devono essere fluidi e progressivi, evitando cambi bruschi.
- **Continuità e coerenza** → Le transizioni devono collegare visivamente i diversi stati dell’interfaccia, garantendo una navigazione chiara e intuitiva.
- **Gerarchia visiva** → Le animazioni devono aiutare l’utente a capire la relazione tra gli elementi, ad esempio evidenziando elementi in primo piano o indicando chiaramente i cambi di stato.
- **Chiarezza e semplicità** → Il movimento deve essere utilizzato per guidare l’attenzione senza appesantire l’esperienza utente.

Nel material design, i widget mostrati a schermo possono reagire, “sollevandosi” quando toccati. Spiegare perché non sono “premuti”, come normalmente accade con oggetti nel mondo reale (es. tasti di una tastiera o pulsante).

Nel **Material Design**, i widget non vengono "premuti" verso l'interno quando toccati, ma si **sollevano**. Questo comportamento è diverso da quello che avviene nel mondo reale con i pulsanti fisici, come quelli di una tastiera, e ha una spiegazione basata sulla **metafora dei materiali (carta e penna)**:

- Quando un elemento viene toccato, l'aumento della sua elevazione (asse Z) porta ad **ombra più marcata** creando un effetto visivo che ne indica l'attivazione.
- Se un elemento si abbassasse, potrebbe confondersi con lo sfondo o altri elementi dell'interfaccia. Il sollevamento invece rende evidente il feedback e aiuta a capire il suo stato attivo in quanto aumentando le dimensioni della propria ombra risulterà decisamente più evidente per l'utente.

Interazione Responsive e Material Design: spiegare cos'è e quali caratteristiche si considerano per la sua progettazione e realizzazione, aiutandosi anche con esempi

L'**interazione responsive** nel Material Design si riferisce alla capacità dell'interfaccia di reagire in modo immediato e naturale agli input dell'utente, fornendo feedback visivi chiari e migliorando l'esperienza d'uso. L'obiettivo è far sentire l'utente coinvolto e soddisfatto, incoraggiandolo a esplorare l'applicazione.
Caratteristiche da considerare nella progettazione:

- **Input dell'utente:** anche se lo schermo è 2D, il movimento e le ombre simulano una profondità 3D, creando un'interfaccia più realistica. Questo aiuta l'utente a percepire gli elementi come **tangibili** e **reattivi**, migliorando la comunicazione tra lui e il sistema.
- **Reazioni della superficie:** il sistema risponde istantaneamente con una conferma visuale nel punto di contatto.
- **Feedback e interazione con gli oggetti:** il materiale può reagire, sollevandosi quando è toccato, per indicare uno stato attivo
- **Azioni radiali:** le azioni devono iniziare dal punto di contatto dell'utente e propagarsi visivamente in modo chiaro.

Icone di Prodotto: cosa sono e cosa indicano le linee guida del Material Design rispetto ai principi di progetto e realizzazione? Fornire un esempio di progettazione

Le **icone di prodotto** nel Material Design sono **espressioni visive** di un marchio, progettate per essere semplici, con colori pieni, amichevoli e in grado di comunicare l'intento del prodotto. Ogni icona deve essere distinta e deve comunicare e riflettere l'identità del marchio.
La progettazione di una icona si ispira alle proprietà del materiale, tattile e fisica

- ogni icona è trattata come si farebbe con la carta ed è rappresentata con elementi grafici semplici
- dev'essere solida, con pieghe pulite ed angoli netti
- le ombre devono essere coerenti e l'illuminazione soffusa.

I dispositivi Android mostrano le icone di prodotto a **48 dp**, pertanto, quando si creano nuove icone, è bene quindi mantenere questa misura.
Per le **keyline** ci sono forme standard predefinite: cerchi, quadrati, rettangoli, linee ortogonali e diagonali.
Le icone vengono costruite attraverso livelli sovrapposti:

1. **finitura:** illuminazione soffusa per un aspetto più realistico

2. **sfondo:** è l'elemento più basso, la base dell'icona
3. **primo piano:** elemento principale dell'icona
4. **colore:** può essere **spot** (applicato ad una piccola porzione) o **flooding** (applicazione all'elemento pieno fino ai lati).
5. **ombra:** dà profondità.

CAPITOLO 12 - ADB e sensori

Cosa vuol dire abilitare l'integrazione con ADB?

L'**Android debug bridge (ADB)** è un fondamentale tool che permette di interagire con un'istanza del dispositivo Android (fisico o emulato) attraverso una modalità client-server, tramite linea di comando. Questo consente agli sviluppatori di eseguire comandi sul dispositivo, installare e disinstallare applicazioni, copiare file, registrare lo schermo e molto altro.
Essendo un'app client/server, include tre componenti:

- **Un client:** che gira sulla macchina di sviluppo ed invia comandi ADB
- **Un server:** che gira in background nella macchina di sviluppo e si occupa di gestire la comunicazione tra il client e il dispositivo/emulatore. Si avvia automaticamente quando viene eseguito un comando ADB.
- **Un demone:** che gira in ciascun emulatore o dispositivo. Riceve i comandi dal server e li esegue sul dispositivo.

Per **abilitare l'integrazione** con ADB su un dispositivo fisico, è necessario attivare l'opzione **Debug USB**, che si trova nelle Opzioni sviluppatore (che dovranno essere attivate), dopodichè si collega il dispositivo al PC tramite cavo USB. Mentre su emulatore è già integrato. In entrambi i casi, per verificare che l'ADB sia attivo si utilizza il comando: **adb devices**, se vedi il dispositivo/emulatore elencato, significa che l'integrazione è attiva.

Quali sono i compiti e quali informazioni fornisce il SensorFramework?

Il **Sensor Framework** di Android consente l'accesso ai sensori disponibili sul dispositivo, che possono essere **hardware** (che misurano direttamente proprietà ambientali) o **software** (che non hanno un corrispettivo fisico e si basano su dati di sensori hardware). Questo framework fa parte della libreria **android.hardware** e fornisce diversi strumenti per interagire con i sensori.
Le **principali operazioni** che è possibile eseguire con il Sensor Framework includono:

- Determinare i sensori disponibili su un dispositivo.
- Determinare le capacità di ciascun sensore, come la portata massima, la risoluzione, il costruttore e i requisiti di alimentazione.
- Acquisire i dati grezzi dai sensori e definire la velocità minima di acquisizione.
- Registrare e rimuovere i listener per monitorare i cambiamenti di stato dei sensori.

Presentare come in Android si utilizzano i sensori, indicando classi e procedura.

In Android, i sensori vengono gestiti tramite il Sensor Framework, che fa parte della libreria android.hardware. Questo framework fornisce le API necessarie per interagire con i sensori disponibili sul dispositivo.

Classi principali del Sensor Framework

- **SensorManager:** gestisce i sensori, consente di elencare i sensori disponibili, registrare e deregistrare i listener e ottenere informazioni sulle capacità dei sensori.
- **Sensor:** rappresenta un sensore specifico e fornisce informazioni sulle sue capacità.
- **SensorEvent:** contiene i dati generati da un sensore durante un evento.
- **SensorEventListener:** interfaccia che riceve notifiche quando i valori del sensore cambiano o quando la precisione del sensore viene aggiornata.

Le API dei sensori vengono utilizzate principalmente per due attività:

- **Identificazione dei sensori disponibili**
  - Permette di verificare la presenza di determinati sensori sul dispositivo.
  - Utile per disattivare funzionalità che dipendono da sensori non presenti o scegliere il sensore migliore tra quelli disponibili.
- **Monitoraggio degli eventi del sensore**
  - Consente di acquisire i dati dei sensori in tempo reale.
  - Un evento si verifica ogni volta che il sensore rileva un cambiamento nei dati monitorati.

Per identificare un sensore si crea un'istanza del SensorManager chiamando getSystemService() e passandogli l'argomento SENSOR_SERVICE:

```
private SensorManager mSensorManager;

mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);

List<Sensor> deviceSensors = mSensorManager.getSensorList(Sensor.TYPE_ALL);
```

Se si desidera ottenere solo un determinato tipo di sensore, è possibile sostituire TYPE_ALL con costanti specifiche come TYPE_GYROSCOPE, TYPE_GRAVITY, TYPE_ACCELEROMETER, ecc.

Il monitoraggio degli eventi di un sensore consiste nel rilevare i cambiamenti nei dati che il sensore sta misurando. Ogni volta che il valore di un sensore cambia, il sistema Android genera un evento e lo invia all'applicazione, che può quindi reagire di conseguenza.

Per farlo, è necessario registrare un listener che rimane in ascolto dei cambiamenti del sensore. Questo listener implementa due metodi principali:

onSensorChanged(SensorEvent event) → viene chiamato ogni volta che il valore del sensore cambia.

onAccuracyChanged(Sensor sensor, int accuracy) → viene chiamato quando l'accuratezza del sensore cambia.

Quando l'applicazione non ha più bisogno del sensore (ad esempio, quando passa in background), è buona pratica rimuovere il listener per evitare un consumo eccessivo di batteria.

Illustrare le operazioni da effettuare per leggere i dati dai sensori

Per leggere i dati dai sensori su un dispositivo Android, è necessario seguire una serie di passaggi utilizzando il Sensor Framework.

1. Ottenere il riferimento al SensorManager nel metodo onCreate() dell'Activity il quale permette di accedere ai sensori disponibili sul dispositivo tramite **getSensorList(Sensor.TYPE_ALL).**
2. Registrare un listener per il sensore desiderato, in modo da ricevere i dati dal sensore. Questo va fatto nel metodo onResume() per assicurarsi che i sensori inizino a inviare dati quando l'app è in primo piano.
3. L'Activity deve implementare l'interfaccia **SensorEventListener**, che richiede l'override di:
    a. **onSensorChanged(SensorEvent event):** viene chiamato ogni volta che ci sono nuove misurazioni dal sensore.
    b. **onAccuracyChanged(Sensor sensor, int accuracy):** viene chiamato quando cambia la precisione del sensore
4. Disattivare il listener quando l'app è in background nel metodo onPause().

Per creare l'app di un contapassi, di quali componenti indispensabili ci sarebbe bisogno e in che modo sarebbero utilizzati?

Per creare un'app di contapassi su Android, sono necessari alcuni componenti essenziali:
SensorManager → Per gestire e accedere ai sensori disponibili nel dispositivo.
Sensor → Il sensore specifico da utilizzare:

- **TYPE_STEP_COUNTER:** Conta il numero totale di passi dall'ultimo riavvio del dispositivo.
- **TYPE_STEP_DETECTOR:** Rileva quando viene effettuato un singolo passo.

SensorEvent → istanzia un evento prodotto dal sensore
SensorEventListener → Per ricevere notifiche quando viene rilevato un passo.
Bisogna, innanzitutto, ottenere l'accesso al sensore contapassi, tramite il SensorManager, utilizzando la costante Sensor.TYPE_STEP_COUNTER. Se il sensore è disponibile, viene registrato un SensorEventListener per ricevere gli aggiornamenti.
Quando il valore del sensore cambia, viene aggiornato il conteggio in onSensorChanged(SensorEvent event).
Il contapassi è un sensore a basso consumo, ma in genere è buona norma disattivare un sensore dopo averlo utilizzato e quando l'applicazione viene messa in onPause, poiché il sistema non li disattiva automaticamente e potrebbero sprecare batteria inutilmente.

Per creare l'app di una bussola, di quali elementi indispensabili ci sarebbe bisogno e in che modo sarebbero utilizzati?

Per realizzare un'app bussola è necessario utilizzare specifici sensori dello smartphone, gestiti tramite il Sensor framework. In particolare, i componenti fondamentali sono: SensorManager, Sensor, SensorEvent e SensorEventListener.
I sensori indispensabili per il funzionamento della bussola sono:

- **Accelerometro (Sensor.TYPE_ACCELEROMETER):** rileva i movimenti lungo gli assi x, y, z, utile per stabilire l'inclinazione del dispositivo.
- **Magnetometro (Sensor.TYPE_MAGNETIC_FIELD):** misura il campo magnetico terrestre, fondamentale per determinare il nord magnetico.
- **Vettore di rotazione (Sensor.TYPE_ROTATION_VECTOR):** semplifica il calcolo dell'orientamento del dispositivo.

Per visualizzare la bussola, si utilizza un'interfaccia grafica composta da una **ImageView**, che rappresenta la bussola con una lancetta, e una **TextView** per mostrare l'azimut (angolo di orientamento). Infine, è necessario implementare SensorEventListener per elaborare i dati raccolti dai sensori e aggiornare l'interfaccia utente in tempo reale.

CAPITOLO 13 - Persistenza ed SQLite

Illustrare brevemente i vari metodi di persistenza possibili in Android

La **persistenza dei dati in Android** è essenziale per garantire che le informazioni rimangano disponibili anche dopo la chiusura dell'app. Esistono diversi metodi per salvare i dati, ognuno con vantaggi e limitazioni specifiche:

- **Shared Preferences:** utile se dobbiamo memorizzare piccole quantità di dati, in quanto consente di salvare dati primitivi con la modalità **chiave-valore** in un file e di recuperarle facilmente.
  Esistono due modi per accedere a SharedPreferences:
  - **getSharedPreferences():** permette di creare o accedere a più file di preferenze condivise, identificati da un nome specifico
  - **getPreferences():** è utile quando si vuole utilizzare un unico file di preferenze per una determinata Activity.

  Per **scrivere un dato** nelle SharedPreferences, bisogna:
  - entrare in modalità "modifica preferenze" con il metodo edit()
  - aggiungere i valori con metodi specifici: putString(), putInt(), etc.
  - applicare le modifiche con commit() o apply().

  La **lettura** avviene tramite la chiave associata al valore e i metodi getString(), getInt(). E' importante notare che SharedPreferences non è adatto per salvare dati sensibili, poiché i file non sono criptati.
- **Storage interno:** i file possono essere memorizzati nella memoria interna. Per default sono privati (può essere scritto e letto solo dall'applicazione che lo ha creato), quando l'app viene disinstallata i file sono rimossi con l'app.
  Per **scrivere** un file nello storage interno bisogna creare lo **stream** e aprire il file con **openFileOutput()** che restituisce un **FileOutputStream** attraverso il quale è possibile scrivere sul file tramite il metodo **write()**. Per **leggere** il contenuto del file è possibile, ad esempio, utilizzare un **BufferedReader** e il metodo **readLine().**
  Lo storage interno è sicuro e crittografato, quindi adatto per dati riservati.
- **Storage esterno:** tutti i dispositivi supportano una memoria esterna dove salvare i file. Tuttavia, si tratta di una memoria non sempre disponibile (scheda SD).

I file salvati in questo spazio sono accessibili da tutte le app e possono essere modificati dall'utente. Ci sono due categorie di file che si possono salvare:
    - **file pubblici:** liberamente disponibili
    - **file privati:** che appartengono all'app.

  Per accedere alla memoria esterna, è necessario richiedere il permesso **WRITE_EXTERNAL_STORAGE** nel file manifest. Tuttavia, prima di usarla è buona norma controllare se lo spazio è disponibile tramite **getExternalStorageState()**.
- **Cache:** per salvare alcuni dati in un file cache (quindi in modo non permanente) si può usare **getCacheDir(),** il quale permette di aprire un file che rappresenta la cartella interna in cui salvare file temporanei. Tuttavia, se il dispositivo ha bisogno di memoria, Android eliminerà questi file per recuperare spazio. Anche in questo caso, i file nella cache sono rimossi quando l'app è disinstallata.

Il Database SQLite

SQLite è un database relazionale ideale per la gestione di dati strutturati. I database creati sono accessibili da tutte le classi dell'app ma non fuori dall'applicazione.
Per creare un nuovo database SQLite, Android raccomanda la creazione di una sottoclasse di **SQLiteOpenHelper** e fare override del metodo onCreate() in cui eseguire le istruzioni per creare le tabelle del database.
Inoltre, l'SDK Android include uno strumento per database, **sqlite3** che permette di navigare il contenuto delle tabelle, eseguire comandi SQL ed altre funzioni (da cmd).
I **vantaggi** nell'utilizzare SQLite sono: che è molto leggero il ché lo rende veloce, è semplice da utilizzare e supporta database che possono essere molto grandi.
I **limiti** sono che:

- Si possono memorizzare al massimo 140 TB
- Non adatto a contesti client-server
- Solo un client può scrivere nel database alla volta, il che lo rende inadatto per applicazioni con alto traffico o multi-utente.

Spiegare cos'è l'SQL helper

Quando si decide di utilizzare SQLite come metodo di persistenza dell'app, è buona pratica creare una **classe helper** che estende **SQLiteOpenHelper** per semplificare le interazioni con il database ed introdurre così un livello di astrazione che fornisca metodi intuitivi, flessibili e robusti per inserire, eliminare e modificare i record del database. Viene, quindi, utilizzato per evitare di avere delle query esplicite nel codice.
L'uso di SQLiteOpenHelper offre diversi benefici:

- Automatizza la creazione del database se non esiste.
- Gestisce gli aggiornamenti dello schema quando cambia la versione del database.
- Fornisce un accesso strutturato ai metodi getWritableDatabase() e getReadableDatabase().
- Migliora la gestione delle risorse ottimizzando l'uso del database all'interno dell'app.

Per **usare** SQLiteOpenHelper, è necessario creare una classe che lo estenda e implementare i seguenti metodi:

- **onCreate(SQLiteDatabase db)** → viene chiamato alla prima creazione del database per eseguire le istruzioni CREATE TABLE.

- **onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)** → viene chiamato quando la versione del database cambia, permettendo di aggiornare lo schema senza perdere dati.
- (Opzionale) **onDowngrade(SQLiteDatabase db, int oldVersion, int newVersion)** → usato per gestire il downgrade della versione del database.

CAPITOLO 14 - I servizi

Spiegare cosa sono e a cosa servono i thread

I thread sono dei processi che possono essere eseguiti in parallelo al programma principale. Con Java si possono realizzare in due modi:

- come sottoclasse di **Thread**
- creando un oggetto che implementa l'interfaccia **Runnable**.

L'oggetto creato deve fare override del metodo **run()** e per avviarlo bisogna richiamare il metodo **start()** sull'oggetto.
I thread servono per eseguire più operazioni contemporaneamente all'interno di un programma, migliorandone la velocità e l'efficienza. Es. Due contatori possono partire contemporaneamente, invece di aspettarsi a vicenda.

Quali sono le differenze principali tra thread e servizi? Spiegare quando usare un approccio, quando l'altro.

I **Service** vengono utilizzati quando è necessario che alcune funzionalità rimangano attive anche al di fuori del ciclo di vita dell'app. Un'applicazione può avviare un servizio che continua a funzionare in background, anche quando l'utente sta utilizzando un'altra app. In sostanza, un Service è un componente che opera in background senza interazione diretta da parte dell'utente (non c'è interfaccia utente).
Tuttavia, se si desidera eseguire operazioni al di fuori del thread principale mentre l'utente sta utilizzando l'applicazione, la scelta migliore potrebbe essere quella di usare un **thread separato**. Ad esempio, se si desidera che la musica venga riprodotta solo quando l'app è in primo piano, si può creare un thread all'interno del metodo onCreate(), avviarlo in onStart() e fermarlo in onStop(). Quindi, se il lavoro in background è necessario solo quando l'utente sta interagendo con l'applicazione, la soluzione più appropriata potrebbe essere quella di gestire un nuovo thread.
Un Service, infatti, gira nel thread principale del processo che lo ospita e non crea un thread separato. Ciò significa che se il servizio esegue operazioni che potrebbero bloccare o richiedere tempo (come operazioni intensive o di I/O), è necessario **avviare un thread separato** per evitare errori come l'ANR (**Application Not Responding**).

Descrivere, se ci sono, le differenze tra l'implementazione di un servizio Start e un servizio Bound.

Servizio started: un'activity lo fa partire chiamando **startService()** ed il servizio gira in background indefinitamente. Quando l'operazione è eseguita, il servizio dovrebbe fermarsi da solo.

Servizio Bound: una applicazione si collega al service con **bindService()**. Offre un'interfaccia client/server che consente di interagire con il servizio. Gira finché l'app è collegata al servizio. Più app si possono collegare, ma quando tutte si sono scollegate il servizio è terminato.
Un servizio può essere implementato in entrambi i modi, l'unica differenza sono i metodi di callback, entrambi chiamati dopo l'onCreate():

- **onStartCommand():** chiamato dal sistema quando un componente richiede l'avvio del servizio chiamando startService(), eseguendo il comando il servizio parte indefinitamente. E' responsabilità del programmatore fermare il servizio chiamando **stopSelf()** oppure **stopService()** quando il servizio ha terminato il suo compito.
- **onBind():** chiamato dal sistema quando un'altra componente vuole collegarsi al servizio chiamando **bindService()**. Bisogna fornire un'interfaccia per consentire al client di comunicare con il servizio ed ottenere un IBinder. Gira finché il client non chiude la connessione con **unBindService()**. Quando tutti i client collegati eseguono il comando unBindService() il servizio è rimosso dal sistema.

CAPITOLO 15 - Le connessioni

Com'è comunemente chiamata la classe che espone le API per scoprire i servizi di una rete? Qual è la portata/quali sono i limiti?

Il Network Service Discovery Manager è la classe che espone le API per scoprire i servizi di una rete. È possibile verificare, ad esempio se un due device sono connessi alla stessa rete. Le api supportano i servizi basati sul dns e la "scoperta" è limitata solo alle reti locali.

Cos'è il NSD?
NSD (Network Service Discovery) è un protocollo che permette di individuare e connettersi ai servizi disponibili all'interno di una rete locale (LAN) senza conoscere preventivamente gli indirizzi IP o le porte dei dispositivi.
In Android, NSD è gestito dalla classe NSDManager, che fornisce un'interfaccia per scoprire e registrare servizi in rete utilizzando il protocollo mDNS (Multicast DNS) e DNS-SD (DNS Service Discovery).

La classe comunemente chiamata per esporre le API che permettono di scoprire i servizi di una rete è NSDManager (Network Service Discovery Manager) in Android.

Portata e limiti
Permette di scoprire i servizi disponibili su una rete locale (LAN) tramite protocolli come mDNS (Multicast DNS) e DNS-SD (DNS Service Discovery).
Utile per trovare stampanti, server di streaming o dispositivi IoT in una rete locale.

Limiti:
Funziona solo su reti locali, non su Internet.
Richiede permessi di rete e non garantisce una scoperta immediata a causa della natura asincrona delle richieste.

Dipende dal supporto dei servizi stessi per mDNS/DNS-SD.

Illustrare i passi da effettuare per instaurare una connessione bluetooth e scambiare dati tra due smartphone

Per prima cosa è necessario inserire il permesso per accedere al bluetooth nel manifest. Fatto questo è necessario seguire le seguenti fasi:

1. **Setup:** verificare che il dispositivo sia abilitato per il bluetooth, se non lo è disabilitare ogni funzionalità dell'app basata su BT e comunicarlo all'utente. Richiedere all'utente di abilitare il BT, in due passi utilizzando BluetoothAdapter: (1) si ottiene il BluetoothAdapter con il metodo .getDefaultAdapter(), il quale restituisce null se il BT non è supportato (2) si abilita il BT. Ci si assicura che il BT sia abilitato chiamando .isEnabled(), mentre per richiedere l'abilitazione si utilizza startActvityForResult() con l'intent ACTION_REQUEST_ENABLE.
2. **Ricerca dispositivi:** ricercare i dispositivi tramite il bluetooth adapter che esegue una scansione dell'area locale per trovare i dispositivi visibili, di default la visibilità è di 2 minuti ma può comunque essere cambiata. Una volta individuato il dispositivo con il quale si vuole comunicare, se è la prima volta che si connette con quel dispositivo è necessario fare l'accoppiamento con esso così da poter avere il suo MAC address.
3. **Connessione:** per creare una connessione tra due dispositivi bisogna implementare il meccanismo server-side e client-side, bisogna quindi determinare chi farà da server e chi farà da client. Per evitare questo si può usare la tecnica del bluetoothSocket che permette di preparare ciascun dispositivo per fungere da server, senza che questo venga determinato a priori. Il dispositivo che farà da server dovrà aprire un BluetoothServerSocket aperto che rimarrà aperto fin quando la sua richiesta non verrà accettata oppure fin quando la richiesta non scadrà. Il BluetoothServerSocket non si deve preoccupare dell'abbinamento perché la connessione RFCOMM aspetta fin quando l'utente non si abbina o rifiuta o la richiesta scade.
4. **Trasferimento**: si ottiene con l'inputStream e l'outputStream e si leggono i dati con read(byte[]) e write(byte[]).

Descrivere i passi da compiere per scambiare informazioni tra due dispositivi in una modalità wireless a scelta. WIFI

Dalla versione 4.0 di Android è possibile connettere due dispositivi tramite wifi senza passare da un access point ma lo si può fare direttamente tramite **wi-fi direct**. Con il wi-fi si possono coprire distanze più lunghe rispetto al bluetooth. I passi da compiere per scambiare informazioni sono:

1. Dichiarare i **permessi nel manifest** e **verificare se il wi-fi è supportato** per quel dispositivo, se non lo è bisogna disabilitare tutte le funzioni che richiedono tale funzionalità
2. Aprire un ServerSocket che aspetta una connessione con un client su una porta specifica e la blocca finché non finisce, da fare in un thread in background
3. Creare un Socket sul client, che usa un indirizzo IP e una porta per collegarsi con il dispositivo server

4. Inviare i dati dal client al server (stream di byte)
5. Una volta che la connessione è stabilita il server può ricevere i dati dal client.

CAPITOLO 16 - Grafica ed animazioni

Cos'è il Canvas?

La grafica può essere disegnata sulla **View** o su **Canvas**. Disegnare sulla View è più semplice, ma meno flessibile. Da usare quando la grafica è semplice e non va aggiornata spesso. Mentre sul Canvas è più complicato, ma consente di creare una grafica più complessa che si aggiorna frequentemente.
Il Canvas in Android è un'interfaccia che rappresenta la superficie su cui si possono realizzare disegni. Il disegno avviene su una **bitmap** sottostante, posizionata nella finestra dell'app.
Per disegnare su un Canvas, è necessario:

- Una bitmap (una matrice di pixel).
- Il Canvas stesso, che contiene le chiamate per aggiornare la bitmap.
- Primitive di disegno come **Rect** e **Path**.
- Un oggetto Paint per impostare colori e stili.

Il Canvas può essere creato direttamente nel metodo **onDraw()** di una View o tramite **SurfaceHolder.lockCanvas().** È consigliato realizzare il disegno finale nel metodo **View.onDraw()** per ottimizzare le performance.

Elencare, dando una minima descrizione, alcune funzioni primitive per disegnare nel Canvas.

Alcune **funzioni primitive** per disegnare nel Canvas sono:

- **drawText** consente di disegnare un testo utilizzando il paint specificato
- **drawPoints** permette di disegnare uno o più punti
- **drawColor** riempie l'intero canvas con il colore indicato
- **drawOval** ha la funzione di disegnare determinati cerchi in base al paint specificato
- **drawBitmap** disegna la bitmap specificata.

Descrizione del package Drawable

Drawable è un package per la grafica 2D. Un oggetto Drawable è un'astrazione di qualcosa su cui si può disegnare.
Per istanziare un Drawable vi sono tre modi

1. usando immagini delle risorse: si usano aggiungendo i file in res/drawable
2. usando un XML che definisce le proprietà dell'oggetto (es. ImageView)
3. usando i normali costruttori della classe.

Quali sono le differenze tra TransitionDrawable, AnimationDrawable e Animation?

TransitionDrawable permette di animare il passaggio tra due View, dove una passa in primo piano e l'altra scompare.
AnimationDrawable permette di animare oggetti drawable in modalità **frame-by-frame**, generalmente tramite xml, specificando per ogni frame qual è il drawable da visualizzare (es: immagine) e per quanto tempo.
Animation è una modalità di animazione introdotta con le api 11 che permette di animare un qualsiasi widget della UI sia in maniera dichiarativa (xml) che programmaticamente.
Permette di gestire varie proprietà, tra cui: traslazione, rotazione, trasparenza e dimensione.
Componenti fondamentali per il suo funzionamento sono:

- **ValueAnimator**, un motore di temporizzazione che gestisce le proprietà da animare per l'intera durata dell'animazione;
- **TimeInterpolator** che definisce come cambiano i valori in funzione del tempo
- **AnimatorUpdateListener** che definisce il metodo **onAnimateUpdate()** chiamato ogni volta che viene creato un nuovo frame.