LAM QUESTIONS

1) Differenza MVC e MVVM (in Android)
Pattern Model View Controller (iOS o Android)
Pattern Model View View Model (Android)
→ Design Pattern Architetturale

In entrambi i casi si parla di design pattern architetturali. In particolare, il pattern Model View Controller, utilizzato sia in iOS che Android, è in iOS un design pattern by construction e va costruito così. Mentre per quanto riguarda Android, le guidelines o "architectural components", sono linee guida architetturali per produrre una buona applicazione, ma non si è forzati a seguirle. Quindi il design pattern architetturale in Android può essere usato come non usato.

Detto questo, il MVC è il pattern dove si ha un Controller, ossia la parte operativa quindi le Activity e il ViewModel, che è la classe centrale che fa da collante tra il Model (tutti in dati che sono persistenti) e la View (i layouts e i dati statici) ed è in grado di referenziare entrambi. Quindi la parte attiva è il Controller.
Mentre, quello che succede nel MVVM invece è che la logica viene separata dall'interfaccia utente e viene seguito un flow che è totalmente a cascata e quindi, introducendo gli observables, la parte attiva è la View.

La differenza chiave sta perciò nella separazione dei componenti in componenti logiche diverse: nel caso del MVC, il fatto che il Controller detenga il pieno controllo di View è Model è ottimo nel caso si abbia bisogno di una centralizzazione della logica, ma rende difficile il testing di parti separate, perché appunto senza un Controller non funziona nulla. Inoltre, se in qualche modo cambiamo la View, dovremo per forza cambiare anche il Controller, poiché è quest'ultimo che controlla tutto.
Nel pattern MVVM, al contrario, l'Activity non è più il Controller, ma è la View. Questo avviene perché l'Activity perde un sacco di responsabilità che vengono date al ViewModel, che fa sempre da collante tra la View ed il Model (sostituisce il Controller), ma in questo caso il flusso dei riferimenti è unidirezionale. Quindi eliminando la View dal MVVM, è possibile testare separatamente le due parti ViewModel e Model: lo possiamo fare poiché il ViewModel non ha alcun riferimento indietro nella View, come invece avveniva per il Controller del MVC.

2) Come si fa a fare inter-process communication in Android? *
La comunicazione fra processi in Android avviene attraverso il meccanismo del message-passing. Infatti, in generale, un thread possiede una coda di messaggi chiamata **"message loop"** ed un messaggio è un oggetto che ha determinati campi (che vengono compressi in un oggetto unico) e che può essere mandato e ricevuto da un thread all'altro tramite la funzione **sendMessage( )**.

Quindi, per implementare il message-passing, per ogni thread, è necessario istanziare il suo handler e referenziare il suo looper. Dove l'handler è l'oggetto associato alla coda di messaggi del thread e che appunto serve a gestire un messaggio appena arriva, chiamando la funzione **handleMessage( )**. Mentre il looper è invece un oggetto che gestisce e referenzia la coda di messaggi.

La comunicazione tra thread avviene perciò a basso livello attraverso l'utilizzo di queste due componenti fondamentali che implementano il message-passing in Android.

3) Come un worker thread può modificare la UI e spiegarne un metodo *

Un worker thread, quindi un thread ovviamente diverso dal main thread, può modificare l'interfaccia utente attraverso 3 diversi modi:
1) l'AsyncTask, ossia una classe che wrappa il thread, tuttavia ora deprecata;
2) gli Observables, quindi con l'utilizzo del Design Pattern Observer, che consiste nell'avere delle variabili osservabili dal main-thread
3) attraverso gli Handlers e i Loopers, ossia le componenti fondamentali che implementano il message-passing in Android.

Quest'ultima è la soluzione più classica, che consiste nello scambio di messaggi tra un thread e un altro attaverso la chiamata alla funzione **sendMessage( )**. Per implementarlo, per ogni thread viene istanziato un handler, ossia un oggetto che serve a gestire un messaggio al suo arrivo attaverso la funzione **handleMessage( )**, e referenziato un looper, che è invece un oggetto che gestisce e referenzia la coda di messaggi (chiamata **"message loop"**) del thread.

La **message loop** dev'essere quindi esplicitamente definita per i worker thread. Sarà quindi necessario istanziare ed assegnare ad un thread una coda di messaggi attraverso la funzione **Looper.prepare( )** dell'oggetto Looper, che è una classe helper.
Successivamente, è necessario creare un nuovo **Handler**, il cui costruttore prende in input il Looper che gli abbiamo assegnato, quindi viene associato al Looper. A questo punto, tutto ciò che verrà fatto dentro la funzione **handleMessage( )** dell'Handler, verrà eseguito dentro a questo determinato thread al momento della ricezione di un messaggio. Al suo interno, perciò, implementiamo il comportamento che dovrà avere questo thread all'arrivo di un messaggio.
Infine, per fare in modo che la coda di messaggi sia pronta a ricevere dei messaggi, chiamiamo la funzione **Looper.loop( )** dell'oggetto Looper.

A livello di codice, inoltre, invece che passare un messaggio e implementare il comportamento del thread all'interno dell'handler, che è il modo di farlo più a basso livello, è possibile fare un handler vuoto e semplicemente postare su questo handler, attraverso il metodo **post( )**, un frammento di codice da eseguire, quindi istanziando un nuovo runnable ad esempio dal main thread per fare eseguire questa porzione di codice ad un worker thread. Questo è possibile perché il thread UI ha una coda di messaggi già implicitamente definita.

Esiste però anche l'**HandlerThread**, che ha già di default un suo Looper ed un suo Handler associati e tutto ciò che dovremmo fare sarà **overridare la funzione handleMessage( )**, quindi determinare semplicemente il comportamento di questo thread all'arrivo di un messaggio.

In questo modo è facile passare dei compiti da un thread all'altro. In particolare, l'utilizzo più frequente di tutto questo è ad esempio, dopo che si è fatta una query al database, quando si vogliono stampare da qualche parte nell'interfaccia grafica i risultati che si sono ottenuti.

4) Differenza tra activity e fragment
Entrambi Activity e Fragment sono delle View, ossia delle aree rettangolari della schermata.

Nello specifico, però, un'**Activity** è una componente dell'architettura software di Android che viene avviata dal dispositivo nel momento in cui viene avviata l'applicazione. È uno **"screen-state"**, ossia uno stato dello schermo, quindi si tratta di una schermata intera dell'applicazione: comprende sia la parte strettamente visuale, ossia il layout che l'Activity disegna, ma include anche tutta una serie di metodi per reagire a degli eventi. Ed è attraverso questi metodi che si caratterizza totalmente il suo comportamento.
Perciò l'Activity non è soltanto la schermata visuale, ma è anche tutto ciò che ci sta dietro dal punto di vista programmatico.

Nell'applicazione si possono avere più activities, quindi più schermate che si avvicendano. Nel momento in cui queste schermate si sostituiscono, Android mantiene in memoria uno stack di activities. Questo serve ovviamente a poter navigare avanti e indietro a piacere, tra le schermate dell'applicazione. Se si ha da qualche parte in memoria un'activity stoppata i cui dati sono mantenuti in memoria all'interno di una pila.

I **Fragment**, che sono stati introdotti in Android 3.0, sono delle sezioni modulari di un'activity. Quindi sono tipi di oggetti simili alle Activity, aventi la stessa funzione, ma con la fondamentale differenza che sono atti ad agire come sub-activity modulari, non sono indipendenti e devono essere ospitati all'interno di un'activity. Infatti, potrebbe essere un'esigenza dell'applicazione quella di strutturare un'activity come una collezione di fragment. In questo modo è possibile non cambiare activity, ma decidere di averne una unica ed utilizzare soltanto dei fragment, che si avvicendano.

Si tratta quindi di moduli che possono essere aggiunti e rimossi e che non hanno soltanto un loro layout, quindi una loro istanza grafica, ma si portano dietro anche tutto il comportamento di questo layout. Il **vantaggio** enorme che danno i fragment è il fatto di poterli riutilizzare: non soltanto la parte di layout, ma è possibile riusare tutto quello che sta dentro al layout anche a livello di comportamento.

I fragment sono una grande risorsa per adattarsi a a diversi tipi di configurazione, senza andare ad alterare il codice.

5) HAL

L'HAL di Android è uno dei livelli inferiori nello stack del software del sistema operativo Android (proprio sopra il kernel di Linux) e si tratta di un'implementazione specifica dell'hardware che fornisce un'interfaccia standard tra i servizi di sistema, quindi le API superiori (es: accedere alla fotocamera, accendere il Bluetooth, ecc.), e i driver hardware dei dispositivi per quel servizio (es: il driver della fotocamera). Viene implementata dai fornitori di hardware, non da Google.

Perciò, mentre il kernel è lo stesso per tutti i dispositivi, l'HAL è propria del singolo dispositivo ed è ciò che devono implementare gli sviluppatori di ogni tipo di telefono per rendersi compatibili con la piattaforma Android, che è agnostica sulle implementazioni di driver di livello inferiore.

6) Come funziona l'aggiornamento del view model all'observer

7) Gerarchia del file system di resources e come Android sceglie quali risorse usare

RESOURCES:
In generale, in un'applicazione viene fatta una separazione tra presentazione dell'app, quindi la parte statica di layout, e gestione dei dati, che viene fatta nella parte programmatica.
L'app si compone quindi di codice e risorse e le risorse sono tutto ciò che non è codice imperativo, come ad esempio i file XML di layout, le immagini, i file audio e video, etc. Le risorse sono ciò che ci fornisce una compatibilità tra dispositivi diversi, configurazioni diverse, versioni diverse del sistema operativo, tra orientamenti del dispositivo (portrait e landscape), tra diverse lingue, etc.

GERARCHIA DEL FILE SYSTEM DI RESOURCES:
Nel momento in cui si va a scrivere un'applicazione, quando necessari, si definiscono una serie di file XML, ognuno per una diversa configurazione, ossia vengono definite in questo modo le risorse.
Ad esempio, se definiamo due diversi file di layout, Android si occuperà di fare il linking tra il

comportamento che abbiamo implementato in Java ed il rispettivo file di layout che ci serve in quel momento. Quindi non c'è bisogno di una ricompilazione, ma bensì verrà fatto un binding tra le due cose a runtime. In gergo questo si chiama **"late binding"**, ossia un legame ritardato, che non avviene quindi a compile time, ma direttamente a runtime.

A livello tecnico, le risorse sono definite all'interno del progetto in una directory chiamata **res**, che sta appunto per ***resources***. La cartella res, a sua volta, conterrà determinati tipi di file XML, divisi in diverse cartelle, tra cui le più importanti sono layout, values e drawable. Dove, quelli che stanno nella directory layout, si riferiranno esclusivamente alla schermata o alla regione di una schermata; quelli che si trovano in variables, descrivono invece valori o variabili che sono statiche nell'applicazione e che verrano recuperate all'occorrenza; mentre nella cartella drawable saranno contenute le immagini in formato .png, .jpg o .gif, quindi dei file BitMap.
È poi sempre possibile accedere ad una risorsa da un'altra risorsa o dal codice Java, in questo caso attraverso la classe R.

COME ANDROID SCEGLIE QUALI RISORSE USARE:
Quindi, le applicazioni Android possono fornire risorse alternative per corrispondere a diverse configurazioni di device. Perciò ogni directory, ossia layout, values e drawable, se si vogliono fornire configurazioni alternative, avrà un suo **alias**, quindi una sua copia seguita da un **qualifier**, ossia un nome che specifica una singola configurazione per la quale le risorse all'interno verranno utilizzate. La directory di una specifica configurazione alternativa, prenderà quindi il nome composto da il **nome della risorsa** ed il **qualifier della configurazione** stessa. Un esempio di questo è la directory ***"values-it"***, che conterrà le variabili e i valori in riferimento alla lingua italiano.

Quando l'applicazione richiede una risorsa per la quale ci sono diverse alternative, Android seleziona quale risorsa usare a runtime, a seconda della configurazione del dispositivo, attraverso un algoritmo. Secondo questo algoritmo, Android:
1) come prima cosa, elimina tutte le risorse, quindi tutti i qualifier, che contraddicono la configurazione attuale del dispositivo
Dopodiché, entra in un ciclo che:
2) scansiona tutti i qualifier ed identifica il prossimo in ordine di importanza (il primo è la lingua, il secondo l'orientamento dello schermo, ecc..)
3) verifica se alcune delle risorse rimaste utilizzano quel qualifier: se nessuna lo utilizza va avanti a scansionare il prossimo qualifier in ordine di importanza, altrimenti elimina tutte le altre risorse che non utilizzano quel qualifier.
Android continua con queste operazioni fino a trovare un'unica directory per la risorsa cercata.

8) Services + differenza tra thread e service
I services sono componenti che girano in background sul main thread e lo mantengono vivo nel lungo tempo. È il duale dell'Activity: un componente ben strutturato, con un ciclo di vita e con una serie di callback, ma senza interfaccia grafica. Inoltre, come l'Activity viene lanciato tramite un intent e dev'essere dichiarato all'interno del **manifest**.

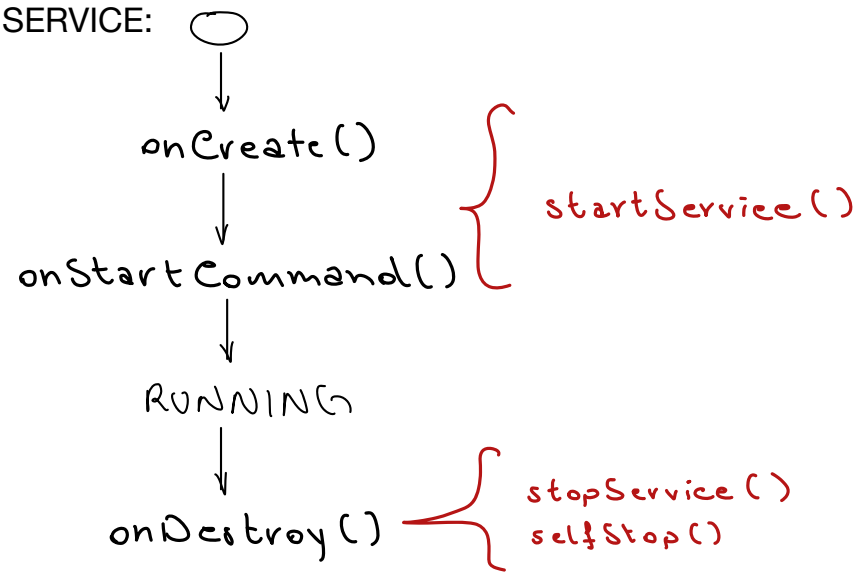
Ovviamente, la presenza di un'activity running e di un service running non sono **mutualmente esclusivi**. Mentre infatti due activity non possono essere running nello stesso momento, un service può invece essere running senza problemi insieme ad altri services o insieme ad altre activity, nonostante giri sul main thread.
Inoltre il service è **indipendente**, perciò se si ha un'Activity che chiama un service è poi quest'Activity muore o va in stop, il service può continuare a girare in background, anche se appunto il componente che lo ha chiamato è stato distrutto.

La sua importante caratteristica è il fatto di poter essere **RUNNING** e continuare a reagire anche quando l'applicazione è chiusa. È l'unica componente che in assenza di un'interfaccia grafica mantiene in vita il main thread, quindi l'applicazione.

Quando il service termina, muore ma muore solamente una componente, non un processo! Quindi un service non è necessariamente associato ad un processo indipendente, sebbene entrambi siano fatti per fare operazioni in background di qualche tipo mentre l'interfaccia utente è impegnata in altro.
Un'altra **somiglianza** che hanno è quella di poter essere killati, quindi terminati, dal S.O. per questioni di memoria, con la differenza che nel caso del service, si ha la possibilità di chiedere al S.O. di riavviarlo subito dopo.

Invece, la **differenza** fondamentale tra un service ed un thread qualunque è il fatto che il service gira sul main thread, mentre il thread no.
Infatti, è necessario usare un thread per fare operazioni che potenzialmente possono bloccare l'interfaccia utente, poiché trasferiamo una determinata quantità di computazione su un thread ausiliario togliendola dal main thread.
Di base, anche se il service fa comunque operazioni di background, non assomiglia in nessun modo ad un thread.

CICLO DI VITA DI UN SERVICE:



I service, inoltre, spesso vengono usati come un'ambiente robusto (poiché girano sul main thread e non vengono killati tanto facilmente dal S.O.) per far girare dei thread.

9) Broadcast Receiver
Il Broadcast Receiver è un componente che viene attivato solamente quando viene scatenato uno specifico evento, come può essere l'arrivo di un SMS, di un'email, la ricevuta di una chiamata, ecc. Serve quindi a far sì che l'applicazione reagisca ad eventi interni o esterni all'applicazione, mettendosi in ascolto passivo di un determinato evento.

Tutte le volte che avviene un evento viene mandato un intent, che in questo caso prende il nome di **"intent broadcast"** ed è generalmente un intent implicito. Infatti il Broadcast Receiver tipicamente si usa per reagire a determinati intent impliciti ed implementare appunto un comportamento di reazione. Può essere paragonato ad un Listener non attaccato ad una View, ma che in generale sta lì e reagisce ad eventi più generali.

Il meccanismo del Broadcast Receiver è quello di crearsi e registrarsi, registrando l'evento al quale si sottoscrive in due possibili punti:

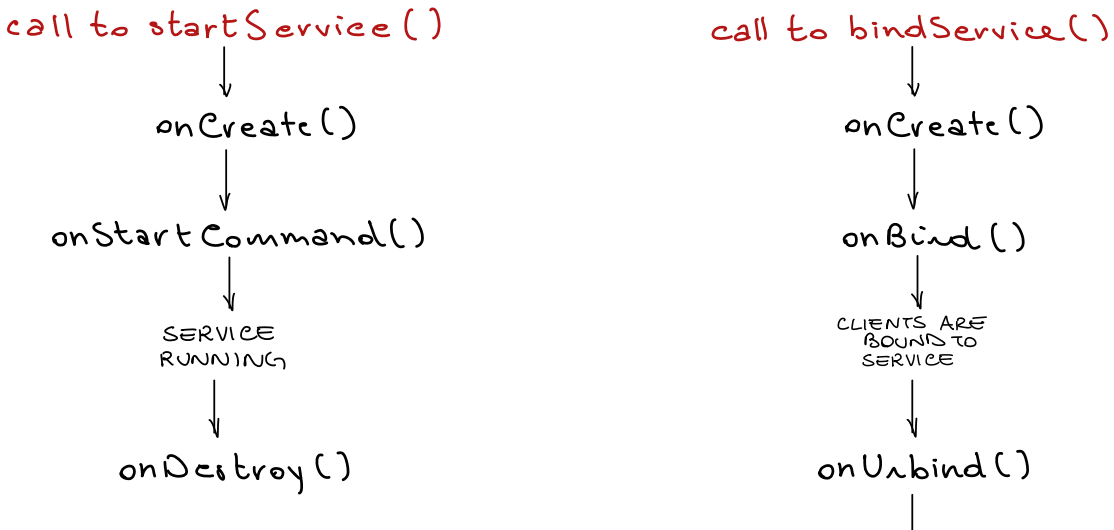
- all'interno del codice XML, quindi sul Manifest, come fanno anche le Activity e i Services, specificando ovviamente anche il suo <intent-filter>, senza il quale è inutile. Registrandolo sul manifest, facciamo sì che il Broadcast Receiver sia sempre attivo, è che quindi possa essere risvegliato anche da applicazione spenta.
- all'interno del codice Java. In questo secondo caso, invece, il Broadcast Receiver viene dichiarato legato al contesto nel quale viene chiamato e registrato, perciò è attivo, per esempio, soltanto mentre l'Activity è attiva.

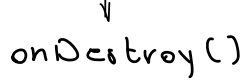
Per quando riguarda, infine, l'handling dell'evento, va implementato un unico punto d'accesso, ossia la **onReceive( )**, che di fatto è l'unico suo stato vivo nel suo ciclo di vita estremamente ridotto:
-> VIENE SVEGLIATO DA UN EVENTO CHE SI VERIFICA (Intent)
-> RACCOGLIE L'INTENT IMPLICITO
-> ESEGUE LA onReceive( )
-> MUORE

10) BoundService
Un BoundService è un particolare tipo di service che ha un binding, ossia un legame, con un altro componente, che può essere il componente che lo ha chiamato, ma anche un componente che ci si è attaccato.
Il **vantaggio** di avere questo legame tra un boundService ed un altro componente, che potrebbe essere per esempio un'Activity, è che un BoundService potrebbe esporre delle funzioni che il componente che è legato a tale service potrebbe voler chiamare. E questo è l'unico modo per farli parlare programmaticamente, senza interpellare il S.O.

Il binding va leggermente a modificare il ciclo di vita del service normale poiché invece di chiamare un service tramite la **startService( )**, lo si fa tramite la **bindService( )**. Quindi il service si avvierà, ma non passerà attraverso la **onStartCommand( )**, ma attraverso un'altra funzione chiamata **onBind( )**, analoga ma che deve ritornare qualcosa, ossia l'oggetto che funge da legame tra il componente chiamante ed il componente chiamato.

È importante il fatto che dal momento in cui un service è bound, quindi dal momento che è stata chiamata almeno una **onBind( )**, non è più la **selfStop( )** che lo fa fermare, ma bensì è la **onUnbind( )** di tutti i componenti che si sono legati al service.
Perciò, il service morirà soltanto quando tutti e due i componenti che hanno chiamato l'**onBind( )** moriranno oppure eseguiranno la **onUnbind( )**. Quindi in pratica questo service dipende dalle componenti che lo hanno chiamato e non è autonomo.
Possiamo dire che, come concetto, è un po' come avere degli hard link ad un file.





11) Foreground Service

Il Foreground Service è un particolare tipo di service che gira in background mantenendo una notifica sempre attiva. Ad esempio, gli aggiornamenti software sono notifiche che non si possono dismettere, ovvero che sono sempre attive.
È inoltre un service più importante di un service in background, poiché è meno probabile che venga ucciso dal S.O. ed essendo che si ha una notifica sempre attiva, si ha un riscontro sul fatto che il service sia attivo o meno.
Ovviamente, servono dei permessi particolari all'app per mostrare dei Service Foreground.

12) Intent

Un intent è un meccanismo per collegare a RUNTIME più componenti nella stessa o in diverse applicazioni, possiamo dire che è come una specie di messaggio contenente un Bundle di informazioni, quindi di dati, utili per il componente ricevente oppure per il S.O. Android. Può perciò chiamare un componente, di solito un'Activity, un Service o un Broadcast Receiver, passandogli eventualmente anche una serie di dati che possono aiutare nella transizione.
C'è bisogno di questo componente poiché per far partire un'Activity, ad esempio, c'è sempre un'intermediazione da parte del S.O, quindi c'è bisogno dell'intent che produce questa richiesta di collegamento.

Gli intent possono essere di due tipi: implici o espliciti. Nel caso di intent esplici sarà necessario specificare il **Component Name**, ossia la destinazione dell'intent, che sarà appunto esplicita. Questo serve a capire quale sarà il componente, quindi l'Activity, il Service o il Broadcast Receiver, più adatto a risolvere questo tipo di richiesta. Nel caso invece di intent impliciti, è il S.O. a scegliere il ricevente a RUNTIME, ossia quando l'intent viene chiamato. Il sistema operativo andrà quindi a verificare se ci sono delle activity della mia o delle altre applicazioni che sono in grado di ricevere questo intent. Questo può avvenire perché nel manifest, le Activity, i Service o i Broadcast Receivers dichiarano anche un **intent filter**, che serve ad esplicitare a quali tipi di intent sono in grado di rispondere questi componenti. In questo secondo caso, il campo Component Name viene lasciato vuoto, mentre quello fondamentale è il campo **Action**. Si tratta di una stringa standardizzata che definisce l'azione che ci si aspetta che implementi il ricevitore di questo intent (come ad esempio fare una foto). È infatti in base a questo campo che il S.O. potrà scegliere i componenti adatti, confrontandolo con l'action nell'intent filter di ognuno. Esempi di azioni più utilizzate sono ad esempio l'ACTION_VIEW, utilizzata quando l'Activity ricevente deve mostrare qualcosa all'utente, è la ACTION_SEND, ossia l'azione che viene chiamata quando l'Activity ricevente è in grado di mandare i dati ricevuti dall'intent, passati nel campo **Extra**, utilizzando dei canali dedicati: ad esempio, l'invio di un'email o di un SMS.

13) Cos'è la sandbox?

14) Content Provider

Il Content Provider è un'interfaccia standard che si implementa sul database o sui dati per rendere disponibili ad altre applicazioni un certo numero di dati della mia applicazione, determinando quali tipi di chiamate le altre applicazioni possono fare. C'è bisogno di questa interfaccia per non dare alle altre app un accesso pieno della mia.
Ad esempio, l'applicazione della galleria lo implementa. Infatti, chiedendo dei permessi, consente di fare un certo numero di query controllate sulla galleria.

15) Paradigma della “Singola sorgente di verità” o Single Source Of Truth

16) Com’è strutturata l’architettura software del sistema operativo Android?

17) Sistema dei permessi di Android con riferimenti alle versioni

18) Ciclo di vita dell’Activity

19) Come funzionano le notifiche in Android?

Le View composte, tipo ListView, RecyclerView, GridView, Spinner, quindi che mostrano un insieme di dati, hanno sempre bisogno di una struttura dati ausiliaria chiamata “**Adapter**” (es. ***ArrayAdapter*** per la ***ListView***), che produce un link tra la ListView come viene mostrata a schermo e la struttura dati che contiene effettivamente i dati da mostrare.
Esiste anche un Adapter fatto per i Cursor —> il CursorAdapter.

Cos’è l’Android SDK?
L’SDK o Software Development Kit consiste in tutta una serie di tool, di strumenti, funzioni, e librerie che servono allo sviluppatore per sviluppare un’applicazione.

Cosa sono le Android API?
Le API di Android sono un’interfaccia tra il sistema operativo e l’applicazione. Quando contattiamo un’API lo facciamo reattivamente, ossia viene fatta una richiesta al S.O., che risponderà quando la risorsa è pronta. Ne sono un esempio i **manager**, ossia classi che wrappano delle funzioni del S.O.

Cosa sono i Fragment e le loro transaction?

Cos’è una View e i modi per specificarla (XML e codice Java)