LABORATORIO DI APPLICAZIONI MOBILI

SDK INSTALL

SDK: software development kit, insieme di tool(strumenti) che aiutano a sviluppare il software(compilatore, tester, debugger, librerie)
IDE: integrated development environment, ambiente grafico da cui i tool sono accessibili(esempio Android Studio)
API: application program interface, insieme di chiamate a cui si sottopone il mondo per interagire con l'applicaizone
Le versioni di API basse hanno un'alta compatibilità ma meno features
Gradle: tool di build per Android. Gestisce le configurazioni di build che non sono più compito dello sviluppatore
CompileSdkVersion: usata da Gradle per compilare i progetti. Dovrebbe essere la più nuova possibile supportata dall'app tra quelle disponibili. E' retrocompatibile
MinSdkVersion: indica qual'è la più vecchia release dell'SDK compatibile con l'app. Es: se un utente ha un cell troppo vecchio l'app non potrà essere installata.
TargetSdkVersion: indica qual'è la nuova release dell'SDK compatibile con l'app, in pratica la versione che ci si aspetta di usare. L'ideale sarebbe uguale alle compileSdkVersion
In breve: minSdkVersion <= targetSdkVersion <= compileSdkVersion
Migliore: minSdkVersion <= targetSdkVersion == compileSdkVersion
Anatomia di un'applicazione:
-Activity: cosa è iniziato
-View: cosa si vede
-Intents: come comunicare
Nella cartella res ci sono le risorse(immagini, layouts, xml files, stringhe)
Nel manifest ci sono le dichiarazioni, i permessi, gli intent filter e i targets
Le app devono essere firmate prima di installarle su un dispositivo reale, serve una chiave prima di creare l'apk e installare l'app sul dispositivo fisico. Per pubblicare bisogna pagare

SYSTEM ARCHITECTURE

Android è una piattaforma basata su Linux per i dispositivi mobili.
Architettura Android:
E' costruita su un **kernel linux** per via della portabilità(facile compilare su differenti architetture hardware), sicurezza(ambiente multi-processo), il runtime si affida al kernel per la gestione dei thread e della memoria
HAL: hardware abstraction layer,i vantaggi sono che nasconde il vero dispositivo, gestisce differenti dispositivi dello stesso tipo, ha delle interfacce standard per esporre le capabilites di livelli inferiori a quelli superiori
Queste interfacce standard vengono implementate perchè Android non sa come sono fatti i driver dei livelli inferiori.
Librerie Native: grafica(Surface Manager), multimedia(Media Framework), database(SQLite), font management(FreeType)
Se si vuole interagire o estendere le librerie native si usa l'Android NDL
ART: Android Runtime, è una virtual machine per compilare che è ottimizzata per i dispositivi con vincoli di memoria e più veloce di quella Oracle

Ha una compilazione Ahead of time ovvero ART compila al tempo di installazione e Just in time dove il codice è parzialmente interpretato quando compilato
APIs: Activity Manager, Packet Manager, Telephony Manager, Location Manager, Notification Manager…
View System(attraverso il quale si costruisce la grafica dell'app), Resource Manager(attraverso il quale si gestiscono le risorse), Notification Manager(attraverso il quale si accedono ai diversi tipi di notifica), Activity Manager(gestisce il ciclo di vita delle activity e fornisce un backstack), Content Providers(per gestire dati tra le app)
Applicazioni: al livello più alto

Componenti Android:
Activity: corrisponde al singolo schermo dell'applicazione, un'applicazione può essere composta da più schermi(attività). Le diverse actvity possono scambiarsi informazioni tra loro. Esse sono composte da componenti grafici(views) e possono interagire con l'utente gestendo eventi. Esse li possono anche generare causati da interazioni dell'utente e devono essere gestiti dallo sviluppatore tramite le callback(es: onClick).
L'Activity Manager è responsabile di creare, distruggere e gestire le activity. Esse possono essere in diversi stati(starting, running, stopped, destroyed, paused). Solo un'activity per volta può essere nel running state.
Intents: messaggi asincroni per attivare i componenti core(esempio le activity). Gli intent espliciti sono quelli dove il componente specifica la destinazione dell'intent mentre quelli impliciti sono quelli dove il componente specifica il tipo di intent, per esempio quando ci sono più scelte per l'utente e vengono presi in carico in base agli intent filters
Servizi: sono come le activity ma eseguono in background e non hanno un'interfaccia grafica. Usati per i compiti non interattivi(networking). Si compongono di 3 stati(starting, running, destroyed)
Content Providers: ogni applicazione android ha il suo set privato di dati(gestito tramite file o db SQLite). Essi sono delle interfacce standard per accedere e condividere dati tra le diverse applicazioni
Broadcast Receivers: un'applicazione può essere raggiunta da eventi esterni. Esempi sono gli SMS, chiamate imminenti….
I permessi dalla versione 6 in poi di Android vengono richiesti a runtime ovvero quando sono usati

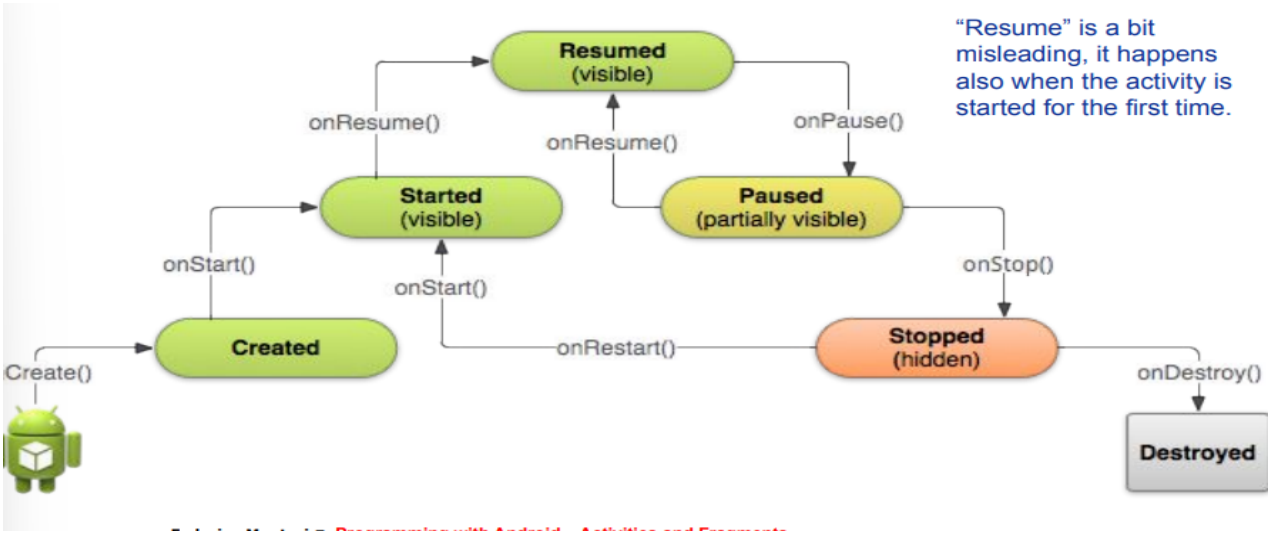
RISORSE

Un'applicazione si compone di codice e risorse dove le risorse sono tutto ciò che non è codice(file di layout XML, immagini, audio…)
Si usano le risorse per separare la parte di layout(presentazione) dalla parte di memorizzazione dei dati e della loro gestione, per fornire risorse alternative su diversi dispositivi.
Il problema principale di Android è che un'applicazione può eseguire su dispositivi diversi tra loro con diverse caratteristiche(size dello schermo, lingua…)
La soluzione è di separare il codice dalle risorse in modo da usare file XML per definirle e prevedere risorse alternative per configurazioni differenti(non fare if-else continuamente)
Costruire il layout, definirne due per due diversi dispositivi, a runtime android individua il dispositivo corrente e carica le risorse appropriate

Le risorse possono essere accedute dal codice usando la classe R(fa da tramite tra Java e le risorse) e conoscendo l'id della risorsa
Si può accedere alla risorse tramite XML(nome del package, tipo e nome) oppure a livello di codice tramite la classe R(generato quando si fa build e fa da collante tra il mondo java e quello delle risorse)
Uno style è un insieme di attributi che può essere applicato a specifici componenti dell'interfaccia o all'intero schermo dell'applicazione(theme). Gli stili possono ereditare proprietà da altri stili
Risorse alternative: risorse alternative per supportare specifiche configurazioni dei dispositivi. A runtime Android individua il dispositivo corrente e carica le risorse appropriate.
Si crea una nuova directory chiamata nomedellarisorsa(nome della directory della risorsa)-qualificatore(nome che specifica una configurazione per il quale le risorse sono usate)
Es: values-it
Quando l'applicazione richiede una risorsa:
-Android sceglie ed elimina i qualifiers che contraddicono la configurazione del dispositivo,per ogni qualifier prosegue iterativamente e identifica il prossimo qualifier nella tabella, se quella risorsa lo usa allora elimina le directory che non lo includono

ACTIVITY

Le activity sono la componente eseguita dal dispositivo e che tiene in piedi l'applicazione, la nostra schermata.
Le activity hanno metodi per reagire a determinati eventi, un'applicazione può essere composta da più activity
Android mantiene uno stack delle activity(quando ci sono applicazioni multischermata), viene fatto un pop dallo stack dell'activity



Stati delle activity:
Quando un'activity viene inizializzata ci sono dei cambi di stato e conseguenti metodi di callback invocati quando una activity cambia stato.
Ciò che cambia tra uno stato e l'altro è in sostanza la visibilità e la capacità di ricevere input dagli utenti
Stati: Resumed, Paused, Stopped
-**onCreate** -> created

Chiamata quando l'activity è creata, contiene le operazioni di inizializzazione(variabili, riferimenti alle view), ha un bundle contenente i dati salvati. Quando termina chiama onstart

-**onStart** -> started(visible)
Chiamata prima che sia visibile all'utente, se ha il focus allora è chiamata onResume altrimenti onStop

Created e started sono talmente veloci che lo stato principale è resumed dove se onResume termina con successo si va in Running

-**onResume** -> resumed(visible)
Chiamata quando l'activity è pronta a ricevere input dagli utenti. Quando termina l'applicazione è in stato running

-**onPause** -> paused(parzialmente visibile, non può ricevere input ed è sovrastata da un'altra activity)
Chiamata quando un'altra activity va in primo piano o quando si preme indietro, per esempio quando un altro componente richede il primo piano
Tutto quello che farebbe passare l'applicazione in onStop passa prima da onPause
Quello che deve fare è fermare tutti i processi che consumano molta CPU
Non salva i dati e non fa operazioni di db

-**onRestart** simile a onCreate ma solo quando l'activity era stata stoppata prima

-**onStop** -> stopped(l'activity è nascosta in background e non può eseguire codice)
L'activity non è più visibile all'utente, per esempio se un'activity sta per essere distrutta e allora serve per fare operazioni di salvataggio
Il sistema mantiene un bundle dello stato di ogni view

-**onDestroy** -> destroyed
L'activity sta per essere distrutta perchè il sistema necessita di spazio in memoria, o qualcuno ha chiamato finish

Se da stopped si chiama il metodo onrestart si va in stato started
Simile a oncreate ma solo quando era stata precedentemente stoppata

Se da paused si chiama onResume si va in stato resumed
Resumed: l'activity è in primo piano e l'utente può interagire
Paused: l'activity è parzialmente coperta da un'altra activity e non può eseguire codice o ricevere input
Stopped: l'activity è nascosta in background e non può eseguire codice

Le activity vanno dichiarate nel manifest
Quando un'activity è distrutta e si naviga indietro il sistema crea una nuova istanza. Si può fare override di onSaveInstanceState prima di essere distrutta, prima di onStop e onrestoreInstanceState quando è di nuovo visibile(nella onCreate)
Tutto ciò perchè quando un'activity è distrutta e si naviga indietro il sistema ricrea una nuova istanza, noi vogliamo tutto com'era e si salva tutto in un bundle instance state

FRAGMENT

Sono una porzione dell'interfaccia utente all'interno di una activity. E' una sezione modulare di una activity(FragmentActivity)

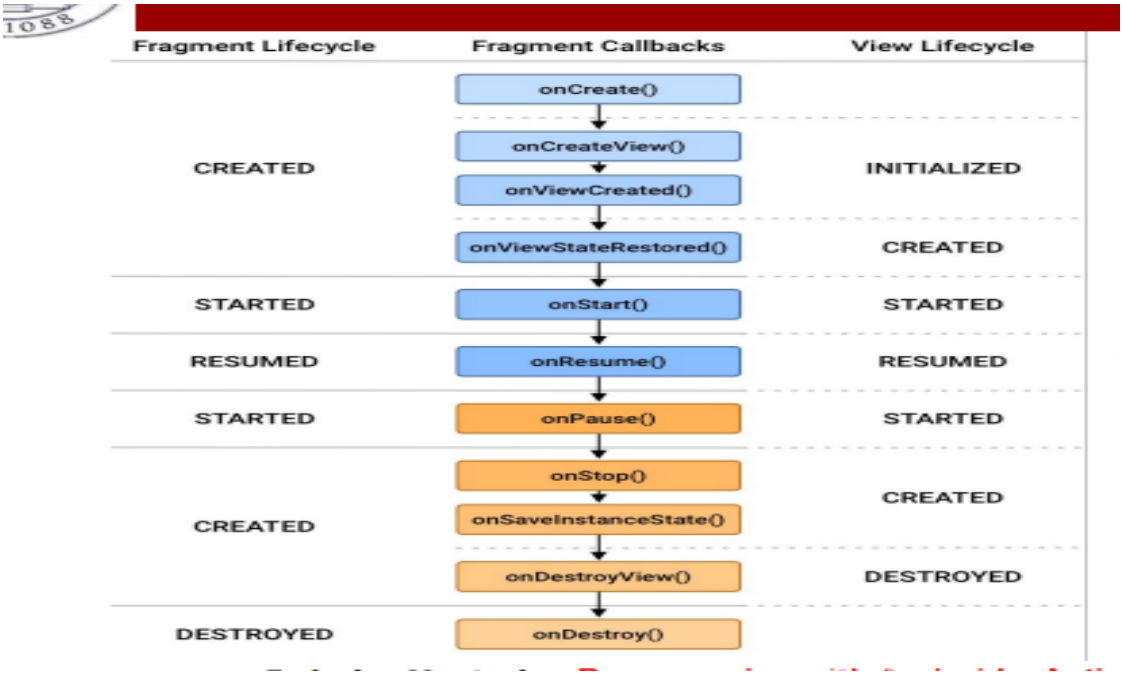
Esempio: strutturare una activity come una collezione di fragment e riutilizzare un fragment in diverse activity

Hanno il loro ciclo di vita parzialmente connesso a quello dell'activity ospitante, il loro layout, possono ricevere eventi in input e possono essere aggiunti o rimossi mentre l'activity sta eseguendo, non possono eseguire da soli

-DialogFragment: migliori dei normali dialog helper perchè si possono riusare

-ListFragment: una buona alternativa per gestire le liste

-PreferenceFragmentCompat: usati per far visualizzare i preference screens



Stati:

-**onCreate**: quando si crea un fragment

-**onCreateView**: quando viene chiamata il fragment disegna la sua area quadrata sullo schermo. Qui bisogna definire il layout che si vuole associare al fragment

E' chiamata quando è ora per il fragment di disegnare l'interfaccia utente per la prima volta(o si torna dal backstack).

Ritorna la view associata all'interfaccia del fragment, il tutto viene fatto tramite un LayoutInflater.

onViewCreated: chiamata per specificare il comportamento del fragment(bottoni….)

quando viene chiamata la onCreateView ci passa dei parametri che possiamo usare per l'inizializzazione(non fondamentali), il layoutInflater è un oggetto che ci da una serie di funzioni per inserire codice xml all'interno di uno spazio(dentro al container mi vai a caricare questo layout)

Il container è il parent ViewGroup del fragment mentre inflate prende la risorsa con cui riempire il container del fragment.

Un'alternativa è passare il layout al super costruttore(meno custom)

onPause: chiamata quando l'utente sta lasciando il fragment

Il ciclo di vita dell'activity nel quale sta il fragment influenza direttamente il fragment
FragmentManager: elemento API che gestisce il ciclo di vita dei fragment, da dentro l'activity lo uso tramite getSupportFragmentManager mentre da dentro il fragment lo uso con getParentFragmentManager.
Con esso gestisco il fragment associato all'activity corrente
Un Fragment può prendere una reference all'activity tramite getActivity, un'Activity può prendere una reference al Fragment tramite getSupportFragmentManager.findFragmentById
Prima che un fragment entri nel ciclo di vita chiama onAttach e alla fine onDetach
FragmentTransactions: i fragment possono essere aggiunti, rimossi, rimpiazzati mentre l'activity sta eseguendo, ogni insieme di cambiamenti che l'activity fa sui suoi fragment è detta transaction.
1)acquisisco un'istanza del fragment manager
2)creo un nuovo fragment e tramite il replace rimpiazzo quello che c'era con il nuovo fragment
3)salvo al backStack e faccio commit(operazioni effettuate effetivamente qui)
se addToBackStack non viene chiamata non posso risalire al fragment che avevo prima altrimenti il fragment viene stoppato e posso ritornare indietro
Sia i fragment che le activity hanno un backstack ma quello delle activity è mantenuto dal sistema mentre quello dei fragment dall'activity ospitante.
Non si può sostituire il fragment che si è dichiarato in maniera statica nell'XML, bisogna fare tutto dinamicamente(esempio usare un frame layout).
FragmentContainerView: è un contenitore di fragment che permette di realizzare dei fragment dinamici(sostituibili), in questo modo tutti i fragment sono dinamici anche se dichiarati nel layout XML, con essa i nuovi fragment possono essere sostituit facilmente. I FragmentContainerView eseguono una fragment transaction quando l'activity inizia.

INTENTS

Intent: oggetto che viene usato dal SO per svegliare un determinato componente, instaura un legame a runtime dove esegue l'associazione tra componenti nella stessa o in diverse applicazioni.
Gli intent possono svegliare solamente Activity, Services e Broadcast Receivers.
Chiamare un componente da un altro componente, passare dati tra componenti(activity, service, broadcast receivers)
Fai questo con questi dati
Si possono pensare come un messaggio con una struttura definita che contiene un bundle di informazioni di interesse per il ricevente e per il sistema
Espliciti: il ricevente è specificato nel component name, usati per lanciare specifiche activity
Con startActivity lo faccio per rimanerci, con startActivityForResult lo faccio aspettandomi di tornare indietro con un risultato
Da quest'anno la startActivityForResult è deprecata(si può usare ma non consigliata)
Impliciti: non so chi raccoglierà l'intent,il ricevente è specificato dal tipo di dati. Il sistema im base ai parametri risvegliail ricevente che corrisponde ai requisiti tra quelli compatibili.
Component Name: componente che deve gestire l'intent(opzionale in quelli impliciti)

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

Le activity possono ritornare risultati, dal lato sender si invoca la startActivityForResult dove si mette l'intent e il requestCode.
Nell'onactivityResult si mette il request code, il result code e i dati ed è invocata quando la seconda activity completa le sue operazioni
Dal lato receiver si invoca getIntent e setResult che verrà inviato solo dopo il metodo finish()

Impliciti: non nominano un target, android controlla quale activity può gestirlo. se almento una viene trovato allora viene fatta partire.

Action: stringa che individua l'azione che deve essere eseguita. Si può usare action per determinare cosa deve fare l'activity chiamata. Action_View è chiamata quando l'activity ricevente mostra qualcosa all'utente mente Action_Send quando l'activity ricevente può inviare dati ricevuti dall'intent
Edit, Main, Pick, View, Search, Send

Data: dati passati dal chiamante al componente chiamato
Category: stringa contenente informazioni sul tipo di componente che deve gestire l'intent
Extra: non sono liberamente indirizzabili dallo sviluppatore
Flags: informazioni addizionali che indicano ad Android come lanciare un'activity e come trattarla dopo averla eseguita
Tramite gli intent filters si può dichiarare che tipo di intent posso gestire

La risoluzione degli intent si basa sul filter, tramite l'**intent filter** nel manifest posso specificare all'applicazione a quali intent sono in grado di rispondere e quale componente è in grado di gestirlo(ogni activity spcificherà intent filter)
Devo passare l'action field test, il category field test, il data field test, se li passa tutti può grstire l'intent
Action test: l'azione specificata deve corrispondere a una di quelle listate nel filter
Category test: ogni categoria nell'intent deve corrispondere a una categoria nel filter
Data test: l'uri nell'intent è comparato alle parti di uri nel filter

Activity Result….

Android Permission System: Permesso di una certa applicazione di accedere ad alcune componenti che di default richiederebbero un permesso
Sopra la versione 6 basta dichiarare i permessi nel manifest, si possono revocare anche in un secondo momento
Inoltre si può verificare se il permesso è concesso(confronto
il valore con permission granted) prima di eseguire un'azione che necessita di permesso. Se non è richiesto si può chiedere il permesso e si attende la risposta dell'utente in modo asincrono

VIEW

La struttura tipo è data dalle activity(schermi), dagli intent(comunicazione tra i componenti), dai layout(posizionamento degli elementi sullo schermo) e dalle views(elementi da posizionare)
Views: blocchi per componenti dell'interfaccia utente
Si possono creare nell'XML o tramite Java

Ogni view può **generare eventi** che possono essere catturati dai listeners
Ogni view ha un focus e una visibilità che possono essere modificate tramite requestFocus() e setVisibility(int)
Gli eventi possono essere gestiti direttamente **dall'XML**(onClick chiama questa funzione), tramite **event handlers**(si estende la classe di quel bottone e si fa overraide di onTouchEvent)
Oppure tramite gli **event listeners** dove ogni listener gestisce un singolo tipo di evento e ha un metodo callback che viene invocato in occorrenza dell'evento
Si possono simulare degli eventi dal codice tramite perform…()
Es: performClick->onClick

LAYOUTS

I ViewGroup sono dei container invisibili che definiscono la struttura di layout per la vista dichiarata all'interno(sono dei view container)
La differenza principale tra un drawable e una view è la reazione agli eventi
I layout sono dei ViewGroup e sono responsabili di posizionare le altre view sullo schermo
Il layout viene caricato quando l'activity viene creata
Ci sono dei layout predefiniti(Linear, Relative, Table, Frame, Constraint)
Linear: dispone le view su una singola riga o colonna dipendente dall'orientamento
Con weight dico quanto spazio devono occupare(quando devono condividere uno spazio)
Con gravity indico l'altezza nello schermo
Relative: dispone le view in accordo al container o alle altre view, l'attributo gravity indica quali view sono più importanti per definire il layout. E' utile allineare le viste in blocchi
Table: layout simile a linear ma viene organizzato tutto in una tabella dove ogni riga è dentro un elemento tableRow
Frame: blocca una porzione dello schermo per ospitare tipicamente solo un oggetto che occupa l'intera dimensione del layout(container per i fragment/immagini)
Absolute: specifica le posizioni degli elementi con x e y
Constraint: l'idea è che definisce i constraint per ogni view, ogni constraint deve essere definito rispetto ad un altra view o ai bordi del padre
Ogni view necessita di almeno un constraint per piano(verticale/orizzontale), aggiungendo 2 constraint opposti la vista sarà nel mezzo
Tutti i constraints devono essere definiti rispetto ad un'altra view già dichiarata
DynamicLayouts: a volte il layout deve essere popolato a runtime dalle viste(ListView, GridView)
Vengono associati ad una struttura dati e disegnati automaticamente in base a quello che c'è in questa struttura dati
Serve però un componente che metta in relazione la struttura dati con il layout dinamico, questi oggetti sono gli andapeter
Con gli adapter possor prendere dati da un'altra sorgente e mapparli come elementi dell'adapterview
AdapterView: i suoi figli sono determinati da un adapter, gli adapter sono usati per visualizzare dati dinamici
Es: creiamo un nuovo adapter di stringhe e gli passiamo in input il contesto(activity), un layout che definisce come va disegnato un elemento della lista e un array di stringhe
Vorrei che tu associassi l'array a una mia listView definita in precedenza, a questo punto setto l'adapter alla listview
Tutte le volte che io faccio i cambi a questa lista, se io modifico dinamicamente l'array allora

non si aggiorna automaticamente la view
Lo faccio io tramite una funzione notifyDataSetChanged() che però ridisegna tutto(non efficiente)
RecyclerView: non ha un modo definito di organizzare gli elementi, miglior gestione degli eventi e separazione tra dati e layout. Per ogni recycler dobbiamo definire un **LayoutManager** che posiziona gli elementi sulla recycler view.
Dobbiamo creare una classe che estende RecyclerView.Adapter ed estende anche RecyclerView.ViewHolder(definisco l'adapter).
Poi bisogna fare override di getItemCount, onCreateViewHolder(cosa succede quando credo l'oggetto, faccio l'inflate dell'xml nel viewholder)), onBindViewHolder(collega i dati appropriati al viewholder, gli da un comportamento)
Tutto ciò permette una gestione personalizzata del contenuto di ogni singolo elemento
CardView: si può costurire una recyclerview di cardview. Utile per raggruppare contenuti collegati alla stessa entità
TextView: riempite con stringhe, non editabili
EditText: simile alle TextView ma editabili dall'utente
AutoCompleteTextView: usata per rendere facile l'inserimento degli utenti, una serie di suggerimenti è data attraverso un adapter
Button: genera eventi come click, long click, drag…
CompundButton: bottoni che hanno stato check/uncheck(checkBox, radioButton dove solo un bottone può essere selezionato all'interno del radio group, toggleButton che regola on e off)
Spinners: menù a tendina dove le entries possono essere prese da un array di stringhe o tramite uno SpinnerAdapter
DatePicker: seleziona la data e posso impostare l'anno di inizio e fine e le data max
ImageView: posso impostare la source(immagine) e settare le dimensioni
Toast: piccolo messaggio sopra l'activity, usato per segnalare all'utente una conferma o piccoli errori. Si può controllare la durata

KOTLIN

Java è stato il linguaggio ufficiale e il più supportato ma non è il più usato, Kotlin l'ha sorpassato
Linguaggio ufficiale per Android nativo dal 2019
Con Java devo dichiarare il tipo mentre qui il compilatore **inferisce il tipo di variabile** ma è statico(es: x=42 e x="c" errore), è cross platform ed è interoperabile con Java(mix)
Var x:Int=42 oppure var x=42
Con **val** dichiaro una costante
Le **variabili** vanno dichiarate con dollaro
Possiede gli operatori aritmetici, logici e di comparazione
L'if è uguale tranne per la forma contratta var y=if(x==42) 1 else 0
Il costrutto case è when(x) in 0..21 -> … , si possono specificare i range tra i puntini
Gli **array**: intArrayOf(1,2,3) dichiaro già gli elementi mentre con intArray è vuoto
Le **liste** sono simili agli arraylists val lista=listOf<String>("one"...) ma sono statiche
Le **mutableList** possono essere modificate con add
Il while e il for sono identici
Null Safety: il programma non va in crash con valori nulli(NullPointerException)
In Kotlin dichiaro un qualcosa e lo inizializzo sempre oppure è esplicitamente null

Ci sono anche le variabili nullable ma il linguaggio è fatto in modo che chiamarle è sicuro e non genera eccezioni var s:String?=”Hello”
Con il null safety posso dichiarare val l=s.length ma se s è null allora anche l sarà null(se nullable)
Per esempio quando richiamo funzioni se qualcosa è null il valore null viene trasferito al risultato
L’operatore **Elvis** permette di definire un comprotamento di default nel caso in cui l’oggetto su cui sto richiamando la funzione sia null val l=s?.length ?:-1
Le funzioni supportano un valore di default se non viene passato e possono estendere le classi primitive(**extension functions**)
Le funzioni di alto livello possono prendere altre funzioni come input
Le funzioni lambda sono funzioni non dichiarate che sono passate direttamente così come sono e usate una volta
Le classi hanno un costruttore primario dentro le tonde
In Kotlin ho un **setter** e un **getter** che posso usare anche non esplicitamente(ne viene istanziato uno invisibile di default)
Volendo li posso overraidare aggiungendo comportamenti
Le **scope functions** permettono quando voglio effettuare più operazioni su un oggetto di non
scriverlo ogni volta ma tramite.apply lo posso omettere
Differenze principali: type inference, nullSafe, setter e getter impliciti

BACKGROUND OPERATION

Tutte quelle operazioni che non avvengono necessariamente con il feedback di un’interfaccia grafica
Notifiche: messaggi dell’applicazione che possono essere solo informativi e mostrati all’utente o informativi ed attivi che cliccandoci sopra è possibile eseguire direttamente delle operazioni
Scrollando la notifica posso rivelare dettagli aggiuntivi su di essa
La notifica ha un’icona per la barra di stato, un titolo e un messaggio e il **Notification Manager** è responsabile per la gestione di esse
Come sono fatte: piccola icona, nome app, tempo, testo opzionale, icona grande opzionale, titolo opzionale
Grouping notification: le notifiche possono essere raggruppate se più di una notifica è richiesta per la stessa app. Inoltre da android 8 le notifiche devono avere un **canale(**priorità, questione…) per permettere all’utente di avere un controllo sulle notifiche che vogliono vedere
-prendo la reference al NotificationManager
-costruisco il messaggio di notifica
-mando la notifica al NM tramite notify
Pending Intent: contenitore per intent(intent che sarà lanciato in seguito da qualcun’altro), definisce quale è il comportamento del mio tocco sulla notifica(apertura di una activity…)
E’ più un container per un intent dove definisco il contesto e il request code per cui deve essere lanciato
Con setContentIntent definisco cosa succede quando clicco sulla notifica mentre se ho un bottone tramite addAction definisco cosa succede quando lo clicco
NotificationChannels: li posso creare e settare tramite createNotificationChannel nel Notification Manager

La parte grafica della notifica gira su un thread diverso dal main thread dell'applicazione
In generale ogni cosa che richiede più di un millisecondo per essere eseguita deve essere eseguita su un thread a parte(non sul main perchè bloccherei l'interfaccia utente)
Per creare un thread posso:
-estendere la classe Thread
-implementare l'interfaccia runnable
-coroutines(Kotlin)

Il main thread è responsabile per disegnare, mettere in coda gli eventi e chiamare callbacks, se si eseguono altre operazioni non è performante
ThreadPool: collezione di thread che eseguono operazioni in parallelo da una coda. I nuovi task eseguono sui thread esistenti quando vannoin attesa
ExecutorService executorService = Executors.newFixedThreadPool(4);
Si possono passare messaggi tra Thread tramite una coda di messaggi(**MessageLoop** che deve essere definito esplicitamente) associata ad ogni thread, un handler del messaggio associato al thread e il messaggio
Prendo il riferimento all'handler del thread 1 e gli invio il messaggio
Il thread 1 lo riceve, l'handler si sveglia con handle message e implementa il comportamento in risposta
Con Looper.loop sono pronto a ricevere messaggi)
Message m = mThread.handler.obtainMessage(); // new message for mHandler
m.arg1 = "Argument for the message";
mThread.handler.sendMessage(m);

AsyncTask sono usati per operazioni corte e i risultati sono pubblicati sull'UI Thread.
Devono essere creati sull'UI thread, eseguiti una volta sola e devono essere cancellati per fermare l'esecuzione
Par: tipi di parametri mandati all'AsyncTask
Prog: tipo di progresso pubblicato durante l'esecuzione
Res: tipo di risultato della computazione
Dopo aver invocato execute il task va in onPreExecute, poi in doInBackground dove viene eseguito, poi onProgressUpdate e poi onPostExecute

I services sono dei componenti che possono eseguire operazioni lunghe in background e non necessitano una interfaccia utente. Forniscono un ambiente robuto dove ospitare i thread separati dell'applicazione. Non sono processi separati, thread separati, non eseguono altro se non onCreate e onStartCommand
Vengono usati perchè i servizi sono l'unico componente che può mantenere vivo il thread principale
Essi iniziano quando viene chiamato startService(intent), i servizi eseguno in background anche se il componente che li ha fatti partire è distrutto
Terminano o con stopSelf o con stopService(intent) ovvero terminati da altri oppure il sistema decide di terminarli

Intent Service: creati per service semplici, non gestiscono richieste multiple contemporaneamente, gestiscono un intent per volta

Foreground Services: servizio che è continuamente attivo nella barra di stato e non è un candidato da essere ucciso in caso di poca memoria. Creo una notifica, chiamo startForeground e poi stopForeground per riportare in background
I servizi possono essere fatti partire con startService o associati ad un componente tramite bindService, i bound services terminano quando tutti i componenti collegati si sciolgono
Tramite l'iBinder il componente può mandare richieste al servizio, l'iBinder viene mandato come parametro nell'OnServiceConnected al componente

Broadcast Receiver: componente che ascolta costantemente e che è attivato solo quando uno specifico evento è occorso. L'evento in questione è un intent implicito. Bisogna registrare il Broadcast Receiver all'evento(tramite registerReceiver) e gestirlo
onReceive viene attivato quando l'evento registrato occorre, quando onReceive finisce esso muore
Dopo aver gestito l'evento il BroadCastReceiver viene distrutto
In seguto posso mandare gli intent gestiti dai Broadcast Receiver tramite sendBroadcast(intent)

DATA MANAGEMENT

Come gestire la persistenza dei dati
Shared Preferences: modo conveniente per memorizzare parametri di configurazione sul disco, strutturati in chiave-valore. Possono essere pubbliche(fino Android 7) ovvero le altre applicazioni possono leggerle potenzialmente o private ovvero ristrette a livello applicazione o activity
Le posso leggere tramite un get
Vengono editate con sharedPreferences editor, bisogna fare commit delle operazioni alla fine. I preference screens sono dei fragment che si legano automaticamente alle shared preferences
getSharedPreferences(String preference, Context.MODE_PRIVATE);
Per modificarle serve uno SharedPreferences.Editor dove bisogna fare commit(blocca il main thread, Apply no) alla fine.
PreferenceScreens: si può interagire con la SharedPreference di default tramite il preference screen. Per riempire lo schermo si estende il PreferenceFragment

FileSystem: dati sul dispositivo non condivisibili(file specifici android in res/raw o res/xml) oppure file esterni condivisibili
I file non condivisibili onboard sono salvati in /data/data/<package>/files e ci scrivo tramite context.getFilesDir
Mentre sulla SD Card sono in ContextCompat.getExternalFilesDir e per scrivere servono i permessi
I file di testo raw non possono essere modificati ma posso estrarre il contenuto con InputStream file = getResources().openRawResource(R.raw.myfile);
Mentre per i file xml uso l'XmlResourceParser

Per condividere dati si può fare con gli intent tramite action_send

SqLite: ogni dispositivo mobile dispone di un db sqllite, database leggero basato su SQL. Serve un database locale perchè non sempre si può assumere di avere connettività.

SSOT ovvero il concetto per la quale quando vado a prendere i dati è buona programmazione prenderli sempre dallo stesso posto
Il db risulterà come una gradeTable con dentro l'id(intero, primary key) e i vari campi
Nella onCreate creo le tabelle "create table"+table_grades+"("+vari campi…
La connessione la chiudo nella onDestroy con sql.close
Delete: db.delete(mySQLiteHelper.TABLE_GRADES, "id = ?", new String[] {Integer.toString(id_to_delete)});
Il secondo parametro è la condizione di where
Update: db.update(mySQLiteHelper.TABLE_GRADES, values, "id = ?", new String[] {Integer.toString(id_to_update)};
Search:

Cursori: i cursori memorizzaziono i dati restituiti da una query al db, in seguito andranno parsati. Hanno i metodi getCount, moveTo, close. Bisogna guardare dentro al cursore per vedere i risultati della query

ContentProviders: sistema per accedere ai dati condivisi. Per ogni content provider sono assegnati uno o più URI dove l'authority e il db e il path è la tabella nel db. Si possono fare query e poi si può navigare il cursore.
Essi sono il metodo delle applicazioni di mettere a disposizione i propri dati, in generale se definisco un content provider voglio dare l'accesso ai miei dati ad altri client
Lato client: invoca il contentResolver e poi fa una query come se fosse un db

NETWORK OPERATIONS

Tutte quelle operazioni riguardati le interazioni con tutto ciò che c'è di remoto fuori dal telefono
Per eseguire delle operazioni in rete bisogna dare permessi nel manifest. Queste operazioni sono costose in termini di batteria, tempo e costi.
App **lazy first**:
Le operazioni di rete sono costose
-ridurre il numero di operazioni ridondanti ovvero se l'app necessita di frequenti aggiornamenti usare la cache per non scaricare ogni volta
-operazioni temporizzate ovvero aspettare ad eseguire operazioni di rete
-raggruppare le operazioni insieme invece di fare operazioni simili in volte diverse

User Preferences: gli utenti vogliono eseguirle solo se connessi al wifi, eseguire le operazioni di sincronizzazione di notte. Le activity che fanno operazioni di rete dichiarano un intent filter manage-network-usage

Prima di eseguire l'applicazione controlla se la connessione attiva è disponibile attraverso il ConnectivityManager con getActiveNetworkInfo e isConnected
Si può inoltre chiedere il tipo di rete, gli stati dettagliati, se è disponibile e se è in roaming

WebView: è una vista che mostra pagine web e include semplici metodi di ricerca in modo che sia in tutto e per tutto un browser
I metodi principali sono loadUrl che carica la pagina HTML all'url e loadData che carica la pagina HTML cotnenuta nei dati

Non ha un navigationButton ma include metodi callback per tornare indietro, avanti, ricaricare e cancellare l'history

Download Manager: permette di fare operazioni di download di file e memorizzarli nel filesystem interno. Va a interagire col system service che gestisce i download http lunghi.
Il client specifica il file da scaricare tramite un URI. Il download è eseguito in background e si riceverà un intent quando il download sarà terminato(broadcast intent perciò usare broadcast receiver)
Costruisco la richiesta passandogli l'URL del file, la metto in una coda e posso sapere lo stato tramite query. Alla fine posso aprire il file tramite openDownloadedFile

HTTPUrlConnection: libreria, le connessioni sono gestite da un thread separato.
Componente per mandare e ricevere dati in streaming sul web.
Si ottiene una reference tramite URL.openConnection e si prepara la richiesta.
per i comandi post bisogna aggiungere una setDoOutpu
Si può leggere la risposta tramite getInputStream
Alla fine si chiude la sessione tramite disconnect

OKHttp: supporta il multiplexing di diverse connessioni sulla stessa socket. Minore latenza e richieste ripetute possono essere memorizzate sulla cache
Da solo gestisce l'asincronicità delle chiamate
Posso fare chiamate sincrone tramite execute o asincrone dove faccio enqueue ed eseguo una callback con onFailure e onResponse

Volley: libreria HTTP più diffusa dove posso gestire connessioni concorrenti e gestire priorità. Ha meccanismi di caching e posso cancellare le richieste e customizzarle
Le richieste vengono create e poi in maniera asincrona si aggiungono ad una requestqueue
Posso aggiungere gli header facendo override del metodo getHeaders

TCP/IP Communication: usa la programmazione basata su socket. A lato server definisco un socket, accetto la connessione e poi scrivo su un DataOutputStream. A lato client identifico il socket, mando un messaggio che la connessione è stata stabilita e poi prendo i dati tramite un DataInputStream

P2P: per via della confidenzialià(direttamente tra dispositivi), velocità(cammino più corto), meno peso sulla rete, resilienza

WIFI Direct: differente dal bluetooth per via dell'efficienza energetica, il range e il rate di scambio dati
Si ottiene un WifiP2pManager, si scoprono i client attorno e ci si connette. Si definiscono dei listener per essere notificati da specifici eventi

COMPONENTS ARCHITECTURE

Col tempo lo sviluppo Android è cambiato, c'era carenza di design patterns, sono cambiati i linguaggi nativi, ci sono le tecnologie ibride e la gestione dell'associazione tra viste e controller è pesante

Per esempio può succedere che vengano eseguite tante operazioni insieme e il sistema deve uccidere alcuni processi per necessità, l'obiettivo è di avere un'architettura solida che disaccoppi i componenti in modo che non dipendano da altri

ModelViewViewModel(MVVM):
Un viewmodel è un componente che memorizza dati sensibili all'interfaccia utente in relazione al ciclo di vita. Aiuta a sopravvivere ai cambi di consigurazione, se l'activity o il fragment sono distrutti e ricreati non c'è bisogno di salavare le istanze ogni volta
Viene separata la proprietà dei dati dal controllo logico
Per crearlo bisogna estendere la classe ViewModel e prendere il singleton nell'activity
Il contesto del viewmodel è quello passato al ViewModelProvider e un viewmodel non fa mai riferimento ad elementi della view ma il riferimento è one-way

Observables: dati live che si basano sul concetto degli Observables. Gli observables sono delle classi di dati che notificano quando ci sono cambiamenti sui dati osservati
Possono facilmente settare o prendere i valori e devono costruire una funzione di callback per i cambiamenti
Si basano inoltre sul concetto di consapevolezza del ciclo di vita(**lifecycle awareness**), si può implementare un observable in relazione al ciclo di vita ed è utile quando il componente ha bisogno di reagire a cambi nel ciclo di vita
Il metodo getLifecycle lo possono implementare i componenti che hanno un ciclo di vita

LiveData: sono degli osservabili lifecycle aware che notificano solo quando sono in stato attivo(resumed o started). Sono utili per le activity e i fragment perchè possono osservare i dati senza preoccuparsi del loro stato
I MutableLiveData possono cambiare(setter), i LiveData no
I live data vengono istanziati nel ViewModel
Posso osservare i livedata nell'onCreate dell'activity tramite .observe
I valori li posso aggiornare con setValue(chiamato dal main thread) e postValue(chiamato da un worker thread)

MVVM: i layout e dati statici sono la view, le activity e il view model sono il controller e la persistenza è il model
Nel **MVC**: il controller è la parte attiva, se viene cambiata la view viene cambiato anche il controller
Nel **MVVM**: la view è la parte attiva dove la logica è separata dall'UI

Database con Room:
Room ha un livello di astrazione sopra SQLite
-La parte database contiene il proprietario del database ed è il punto di accesso principale
-La parte DAO(Data Access Objects) è l'interfaccia con i metodi per accedere al db
-Le entità sono le tabelle del db
Il database deve essere una classe astratta che estende RoomDatabase
Per ogni Entity, Room crea una tabella nel database dove ogni campo è una colonna
I campi delle entity devono essere pubblici o bisogna fornire getter e setter

Posso cambiare i nomi delle entity e delle colonne
Posso inoltre definire campi unique e le foreign key
Le relazioni si definiscono dichiarando l'altra parte all'interno della classe, se uno a molti si mette una lista di elementi dell'altra parte invece di uno.
Se la relazione è molti a molti si specificano due relazioni uno a molti

Per accedere ai dati servono le DAO, Room crea implementazioni di DAO a runtime dove @querytype può essere insert,update,delete,query
Non operare le DAO operation nel main thread(vietato), si usano worker threads

SSOT Model: si assicura che la richiesta per i data sia sempre fatta su una singola sorgente
L'idea è che quando si richiedono dei dati da remoto bisogna sempre salvarli nel proprio database e fornire dei LiveData ritornati dal db in modo che il ViewModel non sappia chi li ha aggiornati

Retrofit: client HTTP per Java type-safe, traduce automaticamente gli oggetti XML e JSON in POJO.
Nelle entità si usa SerializedName per specificare quale nome ha nel dataframe JSON/XML
Poi si setta il Retrofit Client
E poi come per le DAO si creano interfacce per ogni chiamata remota,ogni chiamata ritornerà un oggetto Call che è un'istanza dell'interazione con il server remoto.
Poi infine come per gli altri client(es Volley) si mette in coda la chiamata

Firebase: piattaforma di sviluppo Google che fornisce un backend reattivo per l'app.
E' un database realtime, fatto per le performance, bassa latenza e pochi dati

NAVIGATION

Menu: appare quando l'utente preme il tasto del menù, utili per dare differenti opzioni senza uscire dall'activity corrente
Ci sono due metodi per crearli:
o metterli in un file dentro res/menu e fare inflate nell'activity nel metodo onCreateOptionsMenu()
oppure tramite Java creando il menu dentro l'activity
Snackbar: simile a un toast ma è attaccata alla vista e può ascoltare gli eventi e dichiarare azioni da eseguire. Se attaccata ad un CoordinatorLayout ha altre features come farla scorrere via
Ha anche azioni come aggiungere ulteriori azioni da essere eseguite , fare il rollback delle operazioni
Il tutto viene fatto prima di chiamare show nel setAction
Dialog: usato per interagire con gli utenti, piccoli messaggi con facili risposte
Possono esserci gli AlertDialog, i DatePickerDialog e i TimerPickerDialog

NavigationDrawer: nascosto quando non è in uso, appare quando si scorre a sinistra o cliccando sull'icona in alto a sinistra
Contiene il layout quando il navigation drawer è nascosto(il main layout) e il contenuto del navigation drawer(simile a un menu)
Come gli altri componenti reagisce agli eventi
Toolbar: come il navigation drawer risponde agli eventi di swipe ma dice all'utente che c'è più contenuto da vedere

Per condividere i dati da android 4 si può usare un Action Provider. l'attuale SHARE, una volta attaccato al menu gestisce apparenza e comportamento
Inoltre serve anche lo ShareIntent appropriato

Il navigation graph è una risorsa XML che connette le destinazioni(fragment) attraverso azioni(eventi)
Il tutto ha luogo dentro un NavHostFragment
Quando si specificano le destinazioni si specificano il tipo(fragment, activity, custom class), il layout(nome del file xml), l'id della destinazione, il nome
Inoltre posso specificare la connessione tra due destinazioni con 3 campi che sono il type field contenente Action, il campo id che contiene l'id per l'azione e il campo destinazione che contiene l'id della destinazione
Per eseguire un azione dobbiamo recuperare il NavHostFragment e ottenere una reference al NavControllerce poi richiamare il metodo .navigate

SafeArgs: una volta creato nelle dipendenze si crea una classe per ogni destinazione assicurando il type safety quando si esegue un azione.

MaterialDesign: è un ambiente 3D dove ogni oggetto a x,y,z e spessore 1dp
Ogni oggetto si trova su un livello diverso e nasconde i livelli più bassi.

GEOLOCALIZATION

Geolocalizzazione: identificazione della posizione geografica nel mondo reale dell'utente
E' resa possibile dalla combinazione di hardware ricetrasmettitori e algoritmi di localizzazione(attraverso il GPS, il WIFI, la rete cellulare)
Context Awareness: possibilità per un sistema di eseguire le sue operazioni in relazione al contesto, l'applicazione si comporta in maniera diversa in base al contesto in cui si trova.
I contesti possono essere primari se definiti come dati grezzi(sensori, gps, tempo) o secondari se è stata eseguita una qualche forma di fusione dei dati
Lifecycle di un contesto: apprendimento e derivazione del contesto dai dati estratti, cosa fare con quel contesto, la raccolta di dati può essere fatta leggendo i dati dei sensori, per estrarre il contesto ci sono diverse possibilità come grafi ecc…

IFTTT: if this then that, definire regole per eseguire azioni che dipendono dal contesto
Es: se cerco Milano e sono vicino a milano cerco la città altrimenti se sono vicino a via milano cerco la via

GPS: ogni satellite manda periodicamente la sua posizione e il tempo corrente
Il ricevitore riceve passivamente i dati, calcola il ritardo del segnale e dal ritardo calcola la distanza dal satellite, tramite la distanza calcola la posizione

WIFI: la localizzazione con wifi è eseguita attraverso la triangolazione o attraverso il radio fingerprinting
I telefoni individuano MAC e SSID sui router nei paraggi, gli smartphone fanno una query ai servizi di localizzazione google, Google fornisce suggerimenti sulla posizioone corrente in base alle informazioni memorizzate sulle reti wifi

Mobile: la localizzazione è operata riconoscendo le celle a cui è collegato
In Android quindi per recuperare la posizione corrente si utilizzano i metodi precedenti che informano il LocationManager, quando la posizione cambia il LocationManager informa il LocationListener

Location Permission:
-Access fine location: permette all'app di usare ogni modo per recuperare la posizione
-Access coarse location: permette all'app di usare solo la posizione che arriva dal wifi o mobile
-Access background location
Il **location manager** è un service di android che tiene in memoria la posizione periodicamente
e a tutte le applicazioni che si sono registrate tramite un **location listener** manda un aggiornamento su di essa
Si crea un oggetto LocationListener e si implementano i metodi di callback, si prende una reference al LocationManager e si registra il locationListener e ogni tot
faccio requestLocationUpdates passando i prametri per dire quanto spesso vorrei gli update in modo da ricevere aggiornamenti sulla posizione dal LocationManager
Poi si aggiungono i permessi nel manifest
Per consumare meno energia si ferma l'ascolto quando non è più necessario dall'applicazione

LocationBasedServices: si ottiene il FusedLocationProviderClient, fonde le richieste fatte da diverse applicazioni(fa una sola richiesta per ogni applicazione) e poi le distribuisce. Si prende la posizione tramite getLastLocation e si crea una LocationRequest che viene aggiornata ogni 10 secondi
Alcune applicazione necessitano di un aggiornamento continuo dove aggiornano i dati dell'interfaccia con la posizione

Map-based apps: esiste un'api di google maps che è l'API v2
Creare un api key per usare la libreria di google maps, associa l'account con l'uso di google
maps per quella particolare applicazione
Nel manifest si specifica quale è la api key
Con getMap si genera un oggetto googlemap
Grazie a questa API posso integrare una mappa google nell'applicazione, inserire informazioni sulla mappa e gestire eventi dell'utente
Per inserire una mappa si aggiunge un **MapFragment**(container dell'oggetto GoogleMap) all'activity e poi si prende la reference nell'activity
Posso customizzare la mappa e definire tramite **LatLng** un punto sulla mappa

I **markers** possono essere usati per identificare posizioni su google map, customizzabili con icona, posizione, titolo ed eventi
Gli eventi sono clickEvents, Drag Events and InfoWindowClickEventts

Posso anche gestire gli eventi sulla mappa come click e camera events

Le forme sono usate per identificare sezioni della mappa(polylines che connettono oggetti LatLng, polygons, circles dove bisogna definire anche il raggio e il centro)

GoogleDirectionApi: è un servizio che calcola la direzione tra una sorgente e una destinazione includendo i diversi mezzi di trasporto.
Si usa un servizio online a cui passo una chiamata get
maps.googleapis.com/maps/api/directions/output?parameters
dove i parametri richiesti sono origine, destinazione, sensori e chiave
I parametri opzionali sono il modo ecc..
Viene ritornato un JSON con tutte le info

GeoCoding: tecnica per convertire l'indirizzo in un LatLng point o viceversa
Tramite getFromLocation e getFromLocationName

Geofencing: tecnica che crea dei confini per permettere all'app di sapere all'app quando un utente entra, esce o rimane in una certa area(proximity marketing…)
Combina la posizione con la prossimità
Si prende il Geofencing client, lo si aggiunge ad una lista e tramite un pending intent si lancia l'evento che si vuole
Per salvare la batteria si vogliono tante notifiche e un raggio ampio, i geofences devono essere distrutti quando scadono e si registrano solo al momento del bisogno

SYSTEM SERVICES

Battery Manager: bisogna usare la batteria saggiamente, questo manager viene notificato quando occorrono delle condizioni speciali(connesso, disconnesso, bassa, okay)
Tramite uno sticky intent posso monitorare i dati

AlarmService: lancia un intent nel futuro dove posso settare quando e ogni quanto a livello di intervalli

WorkManager: dichiara quale è l'attività che si può rimandare, una volta eseguito andrà su un thread in background

PeriodicWork: si può schedulare un lavoro periodico

Monitor and Chain: si possono osservare i cambiamenti nel lavoro usando i LiveData ed inoltre si possono collegare i lavori

Sensor Service: il Sensor Manager interagisce con i sensori(accelerometro, giroscopio, barometro…)
Accelerometro: misura l'accelerazione
Giroscopio: misura l'orientamento
Light sensor
Sensore di prossimità: distanza dagli oggetti(per esempio sapere quando è in tasca)
Ogni sensore ha informazioni sul venditore e il tipo
Si implementa un SensorEventListener dove si eseguono i metodi di callback

Virtual Sensor: ci sono in aggiunta ai sensori hardware dei sensori virtuali

I problemi dei sensori potrebbero essere valori sballati, non affidabili e interferiti dall'ambiente
Questi sensori possono essere utili per individuare le attività dell'utente leggendo dati grezzi e usando metodi di machine learning

AudioService: possono selezionare uno stream e controllare il suono, aggiustare il volume ecc.

TelephonyService: interagiscono con le chiamate, posso chiedere al dispositivo informazioni sulla chiamata

SMS Service: manda messaggi di testo, per mandarli si usa sendTextMessage dove ci sono due pendingIntent sent e delivery quando il messaggio è mandato o ricevuto

Connectivity Service: controllano lo stato di rete del dispositivo(wifi, lte ecc..)
Notificano i cambiamenti a livello di connessione

Wifi Service: gestiscono la connessione wifi tramite getWifiState, isWifiEnable, setWifiEnabled
Possono inoltre listare tutte le connessioni, aggiungerle e aggiornarle e scannerizzare per nuove connessioni
Inoltre possono essere notificati da un broadcast intent in caso di cambiamenti nel wifi

IOS

IOS è formato da:
-Core OS(kernel, gestione dell'energia, certificati, filesystem)
-Core services(servici di rete, threading, preferences, SqLite)
-Media(Audio mixing, recordin, video playback)
-Cocoa Touch(multi touch, core motion, localization, camera)

MVC: Model View Controller, si dividono gli oggetti nel programma in 3 campi
Model: quello che l'applicazione è, ma non viene mostrato
Controller: come il model è presentato all'utente(logica UI)
View: …
Il tutto sta nella gestione della comunicazione tra i vari campi
I controller possono sempre parlare direttamente con il model e la view
Il model e la view non possono mai parlare tra di loro
La view può parlare con il suo controller attraverso le outlet
Il controller mette un target su di sè quindi distribuisce un azione alla vista. La view manda l'azione quando qualcosa succede a livello di interfaccia grafica
A volte la view deve sincronizzarsi col controller ed esso si setta da solo come delegato della view
La view non possiede i dati che mostra ma se serve hanno un protocollo per acquisirli
I controller interpretano e formattano le informazioni del model per la view
Il model non può parlare col controller perciò se ha informazioni da aggiornare usa un meccanismo di broadcast dove i controller interessati si collegano