**Activity**: permette all’utente di interagire con la app fornendo una UI

Un’app può includere una o più activity, ciascuna indipendente e potenzialmente attivabile da un’altra app

facilita le interazioni tra sistema e app:

* Tener traccia di quale activity è attualmente visualizzata per garantire che il processo che la esegue non venga bloccato (resumed activity)
* Tener traccia delle activity con cui l’utente ha interagito in precedenza per permettere di tornare indietro (stopped activity)
* Facilitare il ripristino dello stato di una activity quando il processo viene interrotto (destroyed activity)

La main activity è la prima schermata che appare quando l’utente lancia l’app.

Affinchè le activity di un’app vengano utilizzate è necessario registrare le relative info nel file Manifest e gestire il ciclo di vita di una activity in modo appropriato.

Si definisce una classe Activity

***Ciclo di vita di un’activity***

Un’activity può trovarsi un diversi stati, a seconda del comportamento dell’utente, attraverso i metodi di callback il sistema comunica all’activity il cambio di stato, comunicandogli in che stato si trova attualmente. È necessario definire all’interno delle callback il comportamento atteso dell’activity quando l’utente lascia la activity e quando rientra.

Per liberare la memoria il sistema può eliminare un intero processo, mai una sola activity ma tutte le activity di un processo. Se c’è almeno un activity in foreground il processo non viene killato. In questo caso l’utente si aspetta che una volta riavviata ritornino come prima della kill del processo.

Per salvare lo stato: (guardare slide)

* Classe ModelView: classe che mantiene i dati di stato di un’activity o fragment, quando viene distrutta l’activity o il fragmento, l’oggetto modelView rimane vivo quindi i dati di stato si conservano ovvero i dati dell’activity e lo stato dei widget presenti all’interno di essa, per casi più elaborati ma viene salvato in memoria, non nel disco rigido, quindi può essere usato solo per cambi di configurazione ma non per quando viene killato dal sistema perché risiede in memoria e quindi si perde
* Metodo onSaveInstanceState per dati di tipo primitivo o oggetti semplici (String), chiamato in automatico dopo onStop()

Per ripristinare lo stato fare l’override di onRestoreInstanceState che viene chiamato solo nel momento in cui il bundle è != null, altrimenti bisogna aggiungere un controllo nel onCreate

* Memorizzazione locale: memorizzare in maniera persistente sul disco del dispositivo

*Instance state e bundle*

I dati salvati per il ripristino dello stato sono chiamati instance state, una raccolta di coppie chiave-valore memorizzate in un oggetto di tipo bundle.

Il bundle salva sempre in automatico lo stato degli oggetti di tipo View (i widget) definiti nel layout se hanno un id univoco associato.

Organizzazione in memoria delle activity:

un task è un insieme di activity utilizzate per svolgere un compito, ogni task ha uno back stack, le activity vengono posizionate nello stack secondo l’ordine in cui sono state aperte, ogni nuova activity viene posta a quella precedente, quando viene premuto il tasto di back la top activity viene distrutta e viene visualizzata quella subito sotto.

Le activity sotto sono in stato di stopped, l’activity in foreground è in stato di resumed

**Service**: componente che gira in background senza UI

* Started service: avviato da un componente e rimane in vita fino al completamento del suo task

Anche se il componente che lo ha avviato viene chiuso, lui continua a funzionare

* + User-aware: il sistema deve cercare di mantenere il processo in esecuzione (ascolto musica)
  + User-unaware: il sistema ha la libertà nella gestione del processo
* Buond Service: un componente dichiara di voler utilizzare il servizio

Il service fornisce una API ad un altro processo, esiste quindi una dipendenza tra il processo A bound al service nel processo b, vive fin quando un componente è bound ad esso

Si definisce una classe Service (o dalla versione Oreo, JobIntentService)

**Broadcast receiver**: consente al sistema di notificare all’app eventi al di fuori del suo flusso regolare di esecuzione, non hanno un’interfaccia utente ma possono creare status bar notification per avvisare l’utente di un evento, più comunemente però comunica solo con altri componenti.

Molti eventi sono generati dal sistema (batteria scarica, cattura di un’immagine.. ) ma anche le app possono inviare eventi (notificare di aver scaricato alcuni dati richiesti da un’altra app)

Si definisce la classe BroadcastReceiver e ogni event broadcast è della classe Intent

**Content provider**: gestisce i dati di una app, memorizzati in qualsiasi forma

* Attraverso il content provider le app possono interrogare o modificare i dati se il content provider lo consente
* I dati sono identificati da URI
* Sono utilizzati per condividere database di utilità generale presenti nel sistema operativo (contatti, calendario…), per condividere dati tra app o per condividere dati tra più componenti in un’app articolata

Si definisce una classe ContentProvider

**Attivare i componenti**

Intent

Richiesta dello svolgimento di un’azione da parte di altri componenti, l’app invia un messaggio al sistema android contente un intent e il sistema di occupa di comunicare con l’altro componente.

I content provider invece sono attivati dal content resolver.

Tipi di intent:

* Esplicito: messaggio per attivare un componente ben preciso, l’intent include il nome del componente e android inoltra l’intent direttamente al componente specifico, da utilizzare con i Service.
* Implicito: messaggio per attivare uno specifico tipo di componente, specifica una action, android trova il componente appropriato confrontando il contenuto dell’intent con gli intent filter dichiarati nei manifest di altre app del device.

Intent filter: dichiarazioni da parte di un componente che specifica i tipi di intent che il componente desidera ricevere. Se non ne dichiara può essere attivato solo attraverso intent espliciti.

Costruzione di un intent (guardare tabella slide)

***File manifest***

All’interno del file devono essere definiti tutti i componenti, con relativo nome e classe a cui si riferiscono all’interno del progetto, se un componente non è specificato nel manifest non è raggiungibile ne da intent impliciti ne da espliciti.

**Fragment:** permette la creazione di IU dinamiche e riutilizzabili

È una porzione di activity che ha un proprio ciclo di vita, una propria UI, riceve i propri eventi di input, si può aggiungere o rimuovere mentre l’activity è in esecuzione senza distruggere e ricreare la host activity e può essere riutilizzata in contesti differenti. il fragment deve per forza essere legato ad una activity

Tipicamente una app ha una sola activity sul cui layout si alternano più fragment, questo consente anche di visualizzare più fragment affiancati se il dispositivo è grande (tablet) e visualizzarli alternati se il dispositivo è più piccolo (telefono).

Il ciclo di vita è simile a quello delle activity, con le callback. Ogni callback che riceve l’activity viene ricevute anche dai fragment dell’activity. I fragment hanno callback extra specifiche per un fragment.

Differenze nella gestione del back stack, mentre l’activity è inserita automaticamente nel back stack, il fragment deve essere esplicitamente inserito con addToBackStack() durante la transazione che rimuove il fragment.

Per il salvataggio dello stato è lo stesso delle activity.

La classe che estende è Fragment

Dal fragment posso ricavare l’istanza di activity che lo ospita, dall’activity posso ricavare l’istanza di fragment che ospita.

Una activity che ospita fragment deve essere di tipo FragmentActivity

Comunicazione tra fragment-activity e fragment-fragment guardare video fragment 2

**Architettura**

L’activity/Fragment gestiscono la UI, condivide lo stato attraverso il Lifecycle associato, che è osservabile

I LiveData memorizzano i dati del ViewModel e sono osservabili, i Live data osservano il Lifecycle dell’activity/fragment cosi da aggiornare solo quando l’activity/fragment è attivo.

il ViewModel osserva l’activity/fragment per reagire al cambio di stato, comunica con il repository per aggiornare/ottenere i dati

il repository recupera i dati da tutte le fonti possibili, i dati vengono resi disponibili al ViewModel in termini di LiveData.

Room permette l’accesso al database locale, fornisce i dati direttamente in LiveData, mentre retrofit è per l’accesso remotoma bisogna convertire i dati in LiveData.

***Pattern Observer/Observable***

Lifecycle e liveData sono Observable, a ogni oggetto Observable posso aggiungere (attach) o togliere (detach) degli altri oggetti che implementano l’interfaccia Observer.

Quando cambia lo stato di un Observable, l’Observable scorre l’elenco dei suoi Observer e per ognuno fa una notify(), la notify chiamerà il metodo update() ridefinito da ogni Observer che si occuperà di fare l’azione prevista al verificarsi dell’evento.

**Lifecycle-Aware components**

Il lifecycle è una classe associata all’activity/fragment e contiene lo stato dell’activity/fragment,

è observable quindi posso registrare degli LifecycleObserver, interfaccia che ha un metodo per ogni stato in cui si può trovare l’activity/fragment. Al cambio di stato il lifecycle chiama il metodo corrispondente al nuovo stato per tutti i lifecycle observer che ha, passando il lifecycle owner ovvero l’activity/fragment.

LiveData

Per disaccoppiare la UI dal suo stato e aggiornare la UI solo quando l’activity/fragment sono attivi, quindi visibili all’utente. Il liveData è un contenitore di dati che possono essere osservati.

I LiveData notifica ad un Observer un cambiamento del valore solo se il LifeCycleOwner associato è attivo, l’Observer è un oggetto diverso dal LifeCycleOwner ma vuole ricevere le notifiche solo quando il LifeCycleOwner è attivo. Si può registrare anche un Observer da solo, senza indicare il LifeCycleOwner, ciò vuol dire che l’Observer sarà sempre notificato.

Nel momento in cui il LifeCycleOwner passa allo stato di destroyed, l’observer sul LiveData viene automaticamente rimosso.

ViewModel

Il ViewModel mantiene le informazioni di un’activity/fragment anche a fronte di cambi di configurazione, finchè lo scope è in vita, ovvero finchè l’utente non decide di chiuderlo, solo dopo che lo stato passa in destroyed viene cancellato. Le informazioni sono mantenute nei LiveData

A differenza del Bundle, nel ViewModel posso salvare molte informazioni. L’unico svantaggio è che occupa molta ram e quindi rischia di essere distrutta l’intera app se non è in foreground.

Il ViewModel rimane solo per cambi di configurazione mentre l’OnSavedInstanceState (Bundle) può essere usato per activity distrutte e ricreate.

Il ViewModel implementa l’interfaccia DefaultLifecycleObserver.

Il ViewModel viene creato la prima onCreate dell’app e poi se dovesse esserci cambio di configurazione e quindi un altro onCreate il ViewModelProvider restituisce il ViewModel creato prima, solo quando viene distrutta l’activity/fragment smette di esistere.

La comunicazione fra fragment di un’activity è onerosa, i fragment possono condividere un ViewModel della host activity per gestire la comunicazione.

View binding

Associa il file xml con gli oggetti della activity

Data binding

Associa le view della UI con i valori degli oggetti java del view model. Posso tenere sempre aggiornata la UI con i valori salvati nel ViewModel. Molto potente

Postman permette di fare richieste http ai server e ricevere i risultati, per capire se la richiesta funziona e cosa restituisce.