

ANDROID

XML:

linguaggio di markup utilizzato per conservare dati con una struttura.
in android studio l'aspetto della schermata è descritto da un file xml.

Alcuni nomi diversi rispetto a java classico:

il workspace è detto project

i package sono i moduli

Gradle:

è il build automation tool ufficiale di android, serve ad importare librerie e caricare moduli.

nei suoi file di configurazione viene indicato:

- tipo di build (release, debug, ...)
- tipo di prodotto (versione a pagamento, versione free, versione lite, ecc)
- librerie esterne che o sono già in locale o vengono automaticamente scaricate dove:
 - implementation: indica che la libreria va inclusa nel programma, è utile per selezionare una versione specifica della libreria
 - api: indica che la libreria può essere condivisa tra le varie applicazioni sullo stesso dispositivo, non si ha controllo sulla versione
 - testImplementation...
 - ...

compileSdkVersion

versione della libreria

minSdkVersion

versione di android minima sul quale la mia applicazione si può eseguire

targetSdkVersion

versione massima di android sulla quale la mia applicazione si può eseguire

Struttura di un'applicazione:

un'applicazione ha principalmente tre componenti:

- activity: varie schermate della applicazione
- view: cosa vede l'utente
- intent: sistema per comunicare tra le applicazioni

Manifest:

esiste inoltre un file chiamato manifest che indica al sistema operativo che tipo di applicazione è e cosa fa.

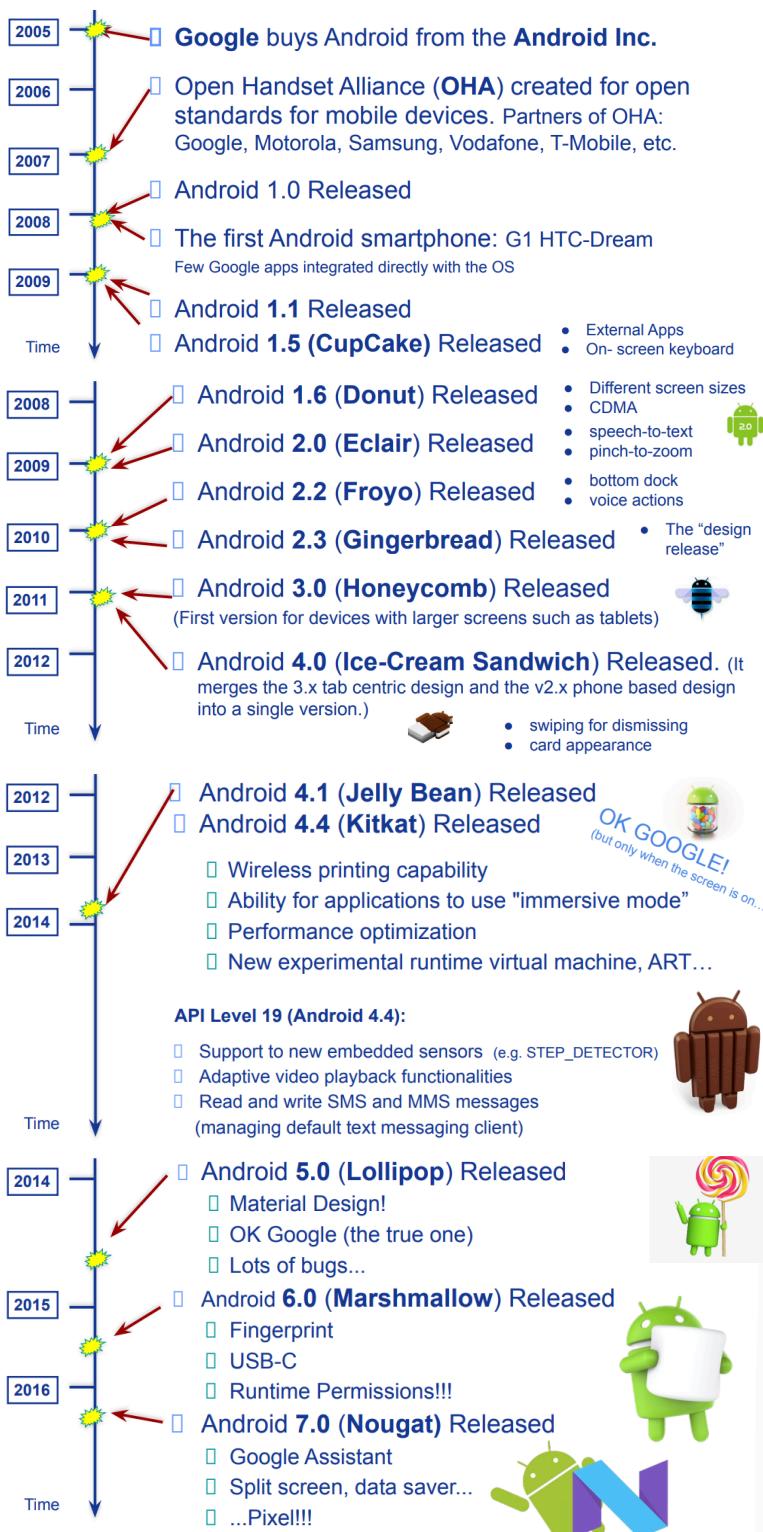
in particolare indica:

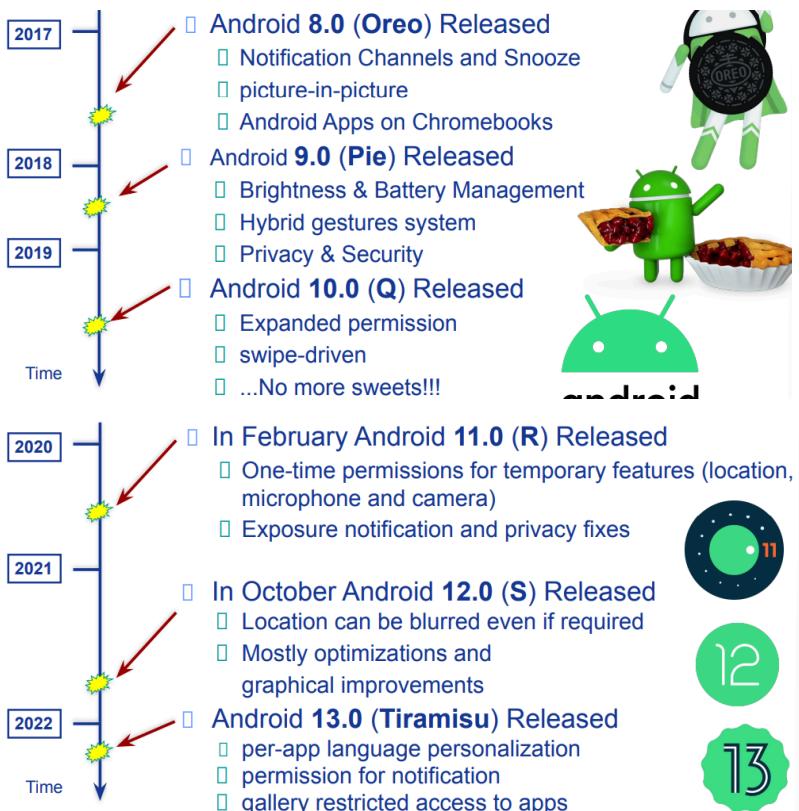
- l'icona dell'app
- le sue activity

- se è chiamabile da altre applicazioni
- a quali intent può rispondere

QUESTA PRIMA PAGINA DI CONFUSIONE è DA SISTEMARE

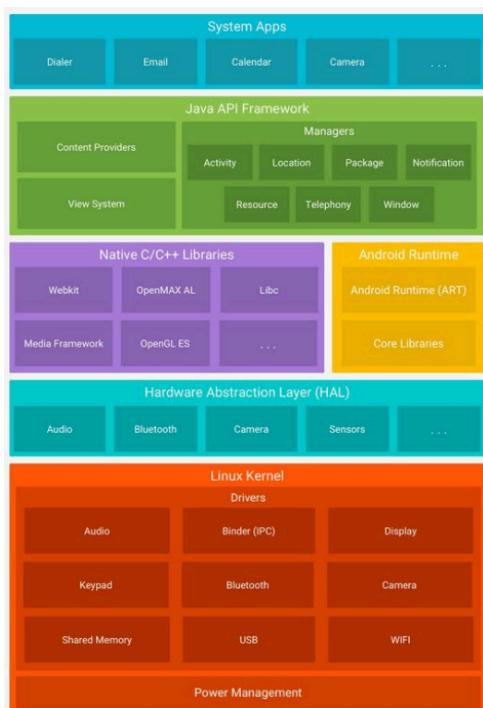
Storia di android:





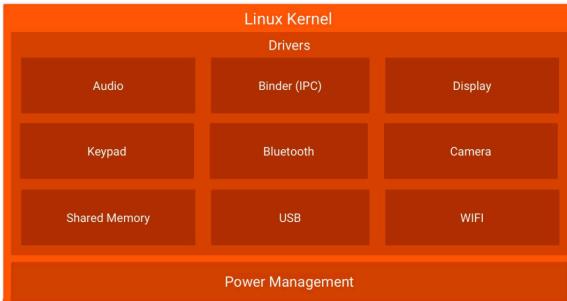
Architettura di android:

android è un sistema operativo nato per dispositivi mobili e basato su linux.
La sua architettura è strutturata a livelli:



Nello specifico:

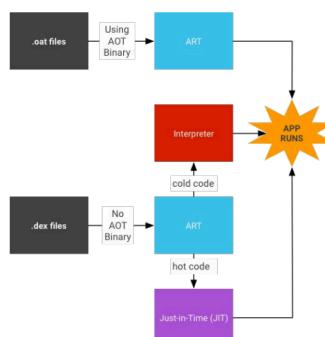
	Applicazioni scritte in java o kotlin
	APIs sono classi globali tramite le quali si interagisce con le funzioni del sistema operativo. Es: activity manager, packet manager, telephony manager, location manager, ecc... le api più importanti sono: <ul style="list-style-type: none"> - View System: Through which you build the APP UI - Resource Manager: Through which you handle resources - Notification Manager: Through which you can access to different kind of notifications - Activity Manager: Which handles the Activity lifecycle and provides a back stack - Content Providers: To share data
	ART (android runtime) virtual machine di android, trattata più nello specifico nel prossimo paragrafo.
	Native libraries codice non java (c/c++) che generalmente lo sviluppatore non modifica (ma si può fare con la NDK).
	HAL sopra al kernel linux si ha la HAL (hardware Abstraction Layer). Questo livello, presente anche in linux, è particolarmente significativo in android in quanto è ciò che mette in comunicazione tutti i tipi di dispositivi diversi (centinaia di migliaia) con il sistema operativo.

	android mette a disposizione questa interfaccia standard ed i vari produttori la implementano.
	<p>Kernel</p> <p>è il livello più basso del sistema operativo. Il kernel è basato su linux ma ha un sistema di sicurezza diverso da quello standard. Esiste un utente root (che NON coincide con l'utente umano) ed ogni app è un utente diverso; questo forza ogni app di accedere unicamente ai propri file. Inizialmente questa cosa era gestita con il meccanismo di permessi di linux, poi c'è stata un'evoluzione fino ad avere una vera e propria sandbox per ogni app.</p>

ART (android runtime):

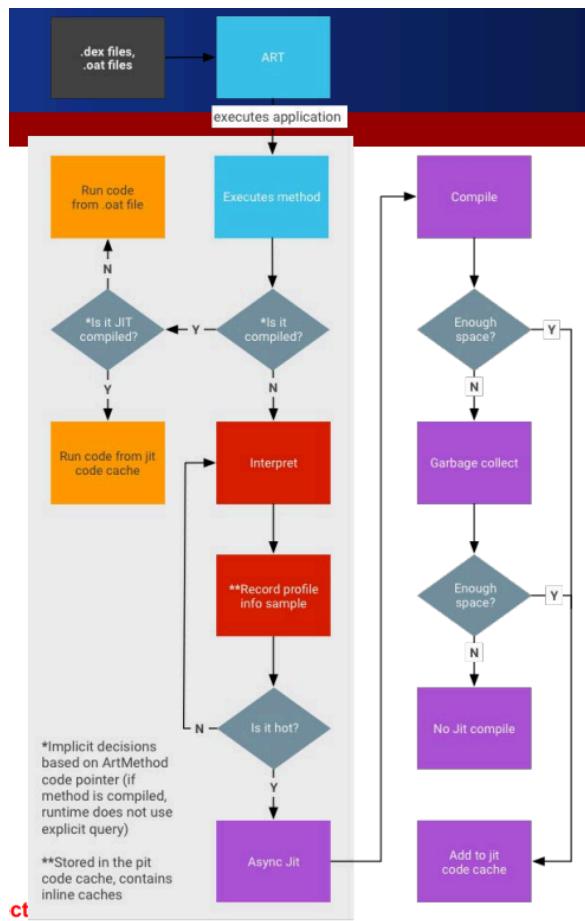
è la virtual machine java di android ottimizzata per dispositivi con memoria limitata. Ha una struttura più complessa della jvm standard:

■ Starting from Android 5.0, ART is used instead of Dalvik
<ul style="list-style-type: none"> □ Several enhancements such as stack size, error handling, AOT... □ more at https://source.android.com/docs/core/runtime/jit-compiler?hl=en
■ Designed to run multiple VM on low end devices
■ Runs DEX bytecode
■ Ahead-of-time (AOT) and Just-in-time (JIT) compilation
<ul style="list-style-type: none"> □ AOT: At <u>install time</u>, ART compiles APPs using an on-device tool called dex2oat → Code compiled at installation □ JIT: code profiling → Code partially interpreted when compiled not available
■ Optimized Garbage collection



viene adottato un approccio ibrido: alcuni file sono già in codice macchina (AOT), mentre altri vengono interpretati a runtime (JIT). I file vengono compilati AOT mentre il dispositivo è inattivo.

quindi il funzionamento della ART è:



Design di un'applicazione android:

la programmazione di un'applicazione android è di tipo reattivo: l'esecuzione avviene tramite la chiamata di funzioni di callback quando avvengono determinati eventi.

Activity:

è una schermata dell'applicazione.

è una componente software che reagisce a degli eventi e che può avere diversi stati (starting, running, stopped, destroyed, paused).

all'apertura dell'app viene attivata e mostrata l'activity Home e solo un'activity alla volta può essere in stato running.

Ospita i vari elementi grafici (chiamati view) dell'app che possono essere istanziati in maniera programmatica (creati dal codice durante l'esecuzione) o dichiarativa (specificati nell'xml dell'app).

le activity possono essere caricate a runtime.

ogni activity ha delle sue funzioni di callback da eseguire quando viene lanciata, chiusa, messa in background, ecc...

View:

le view sono i vari elementi grafici della schermata (caselle di testo, pulsanti, ecc...).

le view possono generare eventi ai quali l'applicazione risponde chiamando delle funzioni di callback.

la maggior parte dell'esecuzione dell'applicazione avviene eseguendo queste funzioni di callback.

Intent:

sono messaggi asincroni utilizzati per attivare componenti di android (come, ma non solo, le activity).

possono essere impliciti o esplicativi:

- impliciti: l'intent è mandato a tutti i componenti di una certa categoria (per esempio tutti quelli che possono far visualizzare un video)
- esplicativi: l'intent ha un destinatario specifico

Servizi (services):

sono componenti che girano sul main thread dell'applicazione simili alle activity ma che servono proprio ad eseguire operazioni (spesso lunghe) quando tutte le activity sono chiuse. Es: il download di un file con l'app chiusa o in background.

Broadcast receiver:

BROADCAST RECEIVER example

```
class WifiReceiver extends BroadcastReceiver {
    public void onReceive(Context c, Intent intent) {
        String s = new StringBuilder();
        wifiList = mainWifi.getScanResults();
        for(int i = 0; i < wifiList.size(); i++){
            s.append(new Integer(i+1).toString() + ".");
            s.append((wifiList.get(i)).toString());
            s.append("\n");
        }
        mainText.setText(s);
    }
}
```

funzione che viene risvegliata all'avvenimento di un evento esterno all'applicazione (ricezione di un sms, connessione ad una nuova rete wifi, ecc...)

Google api:

google mette a disposizione i suoi servizi in maniera molto semplice come integrazione nelle app android.

i due servizi più usati sono google maps e firebase (database remoto non relazionale quindi non SQL, è un database reattivo quindi avverte l'app quando i dati scelti cambiano).

Permessi:

(da non confondere con il sistema di permessi del kernel)

prima di android 6 si dichiarava nel manifest le cose a cui si richiedeva il permesso, il quale veniva richiesto e concesso (o non concesso) all'installazione.

ora i permessi sono richiesti a runtime in base a cosa serve in quel momento e bisogna gestire l'evenienza nella quale i permessi non vengano concessi.

Risorse:

Una risorsa è tutto ciò che in un'applicazione non è codice.

Android divide le applicazioni in codice e risorse per:

- **separare** la presentazione dei dati (layout) dalla gestione dei dati (separation of concerns)
- avere risorse alternative per **diverse configurazioni** di dispositivi
- **minimizzare** la quantità di volte in cui è necessario **ricompilare**.

In android le risorse sono definite tramite file XML ed associate al programma a runtime.

Ogni risorsa è identificata da un **identificatore univoco** (ID) composto dal suo **tipo** (string, color, menu, ecc...) e dal suo **nome** (che o è esplicitamente specificato o è il nome del file senza l'estensione).

Esempio:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Hello world! </string>
    <string name="labelButton">Insert your username </string>
</resources>
```

Resource type
(string)

Questi file si trovano nella cartella “res” e nelle sue sottocartelle (esistono alcune cartelle di default come layout, values, drawable, ecc...).

Più nello specifico:

Resource Type	Resource contained
- res/animator	XML files that define property animations.
- res/anim	XML files that define tween animations.
- res/color	XML files that define a state list of colors.
+ res/drawable	Bitmap files (.png, .9.png, .jpg, .gif) or XML files that are compiled into other resources.
+ res/layout	XML files that define a user interface layout.
+ res/menu	XML files that define application menus.
- res/raw	Arbitrary files to save in their raw form.
+ res/values	XML files that contain simple values, such as strings, integers, array.
- res/xml	Arbitrary XML files.

Da **java** per accedere alle risorse si utilizza la **classe R**, la quale ha un insieme di variabili contenenti l'indirizzo delle risorse.

```

public final class R {
    public static final class string {
        public static final int hello=0x7f040001;
        public static final int label1=0x7f040005;
    }
}

```

R contains
resource IDs
for all the
resources in the
res/ directory.

Quindi per accedere ad una risorsa si utilizza un predicato del tipo:

[<package_name>.]R.<resource_type>.<resource_name>

- <package_name> is the name of the package in which the resource is located (not required when referencing resources from the same package)
- <resource_type> is the R subclass for the resource type
- <resource_name> is either the resource filename without the extension or the android:name attribute value in the XML element.

Esempio:

```

...
final String hello=getResources().getString(R.string.hello);
final String label=getResources().getString(R.string.labelButton);
Log.i(STRING_TAG," String1 " + hello);
Log.i(STRING_TAG," String2 " + label);
...
...

```

Per accedere invece ad una risorsa direttamente da un altro **file XML** si utilizza una stringa del tipo:

@[<package_name>:]<resource_type>/<resource_name>

- <package_name> is the name of the package in which the resource is located (not required when referencing resources from the same package)
- <resource_type> is the the name of the resource type
- <resource_name> is either the resource filename without the extension or the android:name attribute value in the XML element.

Esempio:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="opaque_red">#f00</color>
    <string name="labelButton"> Submit </string>
    <string name="labelText"> Hello world! </string>
</resources>
```

STRING.XML

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <Textview android:id="@+id/label1" android:text="@string/labelText"
    android:textcolor="@color/opaque_red">
        </Textview>
    <Button android:id="@+id/button1" android:text="@string/labelButton">
        </Button>
</resources>
```

MAIN.XML

Tipi di risorse:

le risorse possono essere dei seguenti tipi:

Resource Type	File	Java constant	XML tag	Description
string	Any file in the res/values/	R.string.<key>	<string>	String value associated to a key.
integer	Any file in the res/values/	R.integer.<key>	<integer>	Integer value associated to a key.
array	Any file in the res/values/	R.array.<key>	<string-array> <item> <item> </string-array>	Array of strings. Each element is a described by an <item>
array	Any file in the res/values/	R.array.<key>	<integer-array> <item> <item> </integer-array>	Array of integers. Each element is a described by an <item>

layout	Any file in the res/layout/	R.layout.<key>	<layout>	Defines a layout of the screen
animation	Any file in the res/animator/	R.animator.<key>	<animator>	Defines a property animation (not the only method!)
menu	Any file in the res/menu/	R.menu.<key>	<menu>	User-defined menus with multiple options
color	Any file in the res/values/	R.color.<key>	<color>	Definition of colors used in the GUI
dimension	Any file in the res/values/	R.dimen.<key>	<dimen>	Dimension units of the GUI components
style/theme	Any file in the res/values/	R.style.<key>	<style>	Themes and styles used by applications or by components
drawable	Any file in the res/drawable/	R.drawable.<key>	<drawable>	Images and everything that can be drawn
xml	Any file in the res/xml/	R.xml.<key>	<xml>	User-specific XML file with name equal to key
raw	Any file in the res/raw/	R.raw.<key>	<raw>	Raw resources, accessible through the R class but not optimized

Per quanto riguarda **dimension** (le dimensioni) si hanno varie unità di misura, noi in questo corso utilizziamo i dp ovvero i pixel indipendenti dalla densità dello schermo.

Gli sp sono la stessa cosa utilizzata per il testo.

Per quanto riguarda i **drawable** sono o immagini (quindi si punta direttamente ad un file) o file xml (che contengono a

Code	Description
px	Pixel units
in	Inch units
mm	Millimeter units
pt	Points of 1/72 inch
dp	Abstract unit, independent from pixel density of a display
sp	Abstract unit, independent from pixel density of a display (font)

Drawable type	Description
BitMap File	A bitMap Graphic file (.png, .gif, .jpeg)
Nine-Patch File	A PNG file with stretchable regions to allow resizing
Layer List	A Drawable managing an array of other drawables
State List	A Drawable that references different graphics based on the states
Level List	An XML managing alternate Drawables. Each assigned to a value
Transition	A Drawable that can cross-fade between two Drawable
Inset	A Drawable that insets another Drawable by a specific distance
Clip	A Drawable that clips another Drawable based on its current level
Scale	A Drawable that changes the size of another Drawable
Shape	An XML file that defines a geometric shape, colors and gradients

loro volta dei file oppure che generano direttamente una bitmap).

I **file raw** sono file di altro tipo e sono trattati come stream di byte (quindi non hanno ottimizzazioni specifiche per il tipo di file).

Si salvano come raw i file audio, video, ecc...

Risorse alternative:

Ogni cartella in res può essere ripetuta (con contenuto identico nella forma) aggiungendo in coda un **qualifier** che specifica in quale situazione va utilizzata quella cartella.

I vari tipi di qualifier che si possono utilizzare sono:



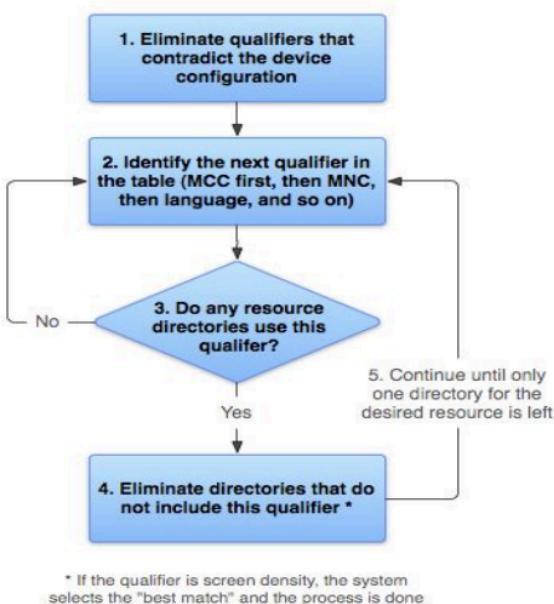
Configuration	Values Example	Description
MCC and MNC	mcc310, mcc208, etc	mobile country code (MCC)
Language and region	en, fr, en-rUS, etc	ISO 639-1 language code
smallestWidth	sw320dp, etc	shortest dimension of screen
Available width	w720dp, w320dp, etc	minimum available module width
Available height	h720dp, etc	minimum available module height
Screen size	small, normal, large, ...	screen size
Screen aspect	long, notlong	aspect ratio of the screen
Screen orientation	port, land	screen orientation (can change!)
Screen pixel density (dpi)	ldpi, mdpi, hdpi	screen pixel density
Keyboard availability	keysexposed, etc	type of keyword
Primary text input method	nokeys, qwerty	availability of qwerty keyboard
Navigation key availability	navexposed, etc	navigation keys of the application
Platform Version (API level)	v3, v4, v7, etc	API supported by the device

Android all'esecuzione elimina tutte le cartelle che non corrispondono alla configurazione attuale.

Poi se ci sono ancora cartelle duplicate android considera quelle con gli attributi più importanti, gli attributi sono riportati in ordine di importanza nella tabella sopra (quindi per esempio language è più importante di screen size).

Se il file cercato non si trova nella cartella più importante viene fatta una ricerca nelle cartelle via via meno importanti.

Esempio:



DEVICE CONFIGURATION

Locale = it Screen orientation = port
 Screen pixel density = hdpi
 Touchscreen type = notouch
 Primary text input method = 12key

~~drawable/~~
~~drawable-it/~~
~~drawable-fr-rCA/~~
drawable-it-port/
~~drawable-it-notouch-12key/~~
~~drawable-port-hdpi/~~
~~drawable-land-notouch-12key/~~

Altro esempio:

Volendo ottenere un'applicazione con una splash screen che si comporta in questa maniera:

Location / Language	US	France	Canada	Italy	Germany	Rest of the world
English	Hello		Hello			Hello
French		Bonjour	Bonjour			Bonjour
Italian				Ciao		Ciao
German					Hallo	Hallo
Rest of the languages						Hello

Si deve fare:

res/	values	Hello
	values-it	Ciao
	values-fr	Bonjour
	values-de	Hallo
	drawable	
	drawable-en-rUS	
	drawable-en-rCA	
	drawable-it-rIT	
	drawable-de-rDE	
	drawable-fr-rFR	
	drawable-fr-rCA	

Activity:

Un'activity corrisponde ad una specifica schermata dell'applicazione.

Android mantiene per ogni applicazione uno stack di activity, le quali si possono trovare in diversi stati.

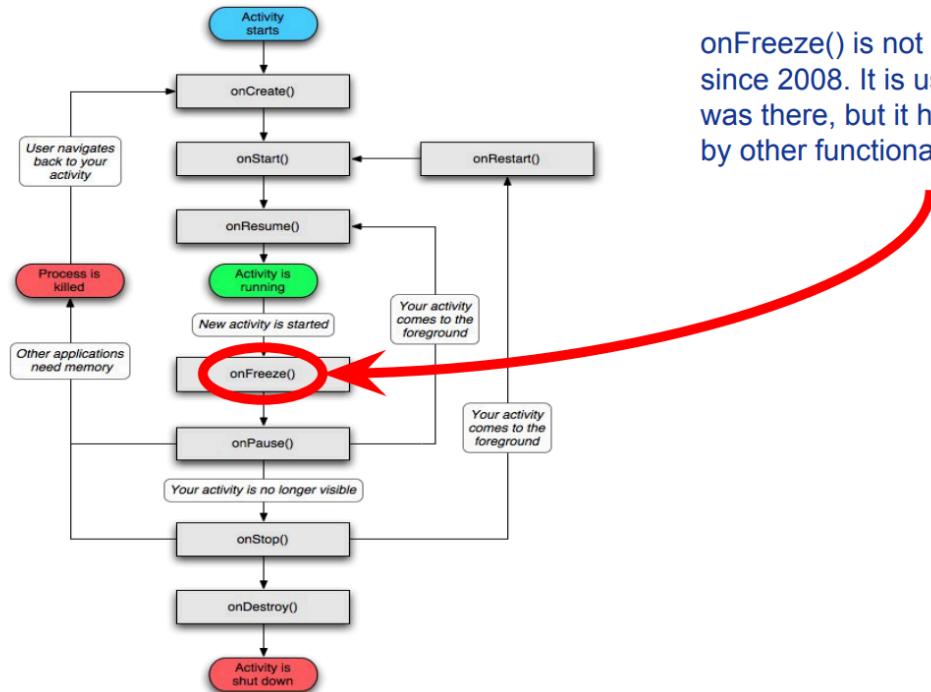
Per dichiarare un'activity nell'xml si fa:

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

Why “MAIN” and “LAUNCHER”?

To show the application in the menu

Ciclo di vita di un'activity:



onFreeze() is not used anymore since 2008. It is useful to know it was there, but it has been replaced by other functionalities.

onCreate() è chiamato quando l'activity viene creata e contiene tutte le operazioni di inizializzazione, inoltre ha un parametro Bundle che contiene i dati salvati relativi a quell'activity.

Quando termina chiama a sua volta **onStart()**.

onStart() è chiamato subito prima che l'activity sia visibile all'utente e se c'è il focus su di essa chiama **onResume()**, altrimenti chiama **onStop()**.

onResume() ha un nome un po' fuorviante in quanto non viene richiamata unicamente quando l'activity ritorna attiva, ma viene chiamata in generale quando è pronta per ricevere input dall'utente (quindi anche subito dopo la creazione se il focus è rimasto su di essa). Se termina correttamente l'activity si trova in stato RUNNING.

onPause() è chiamato quando l'activity passa in background o quando un'altra passa in primo piano (foreground).

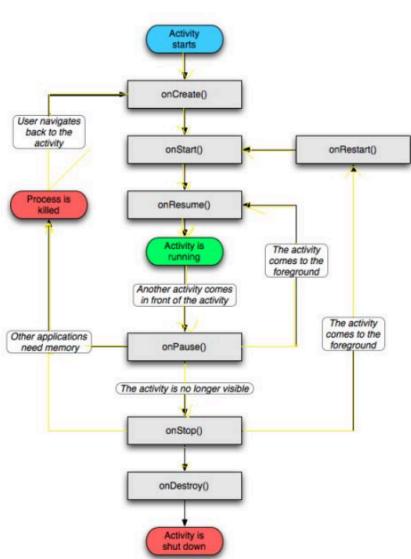
Il suo compito è quello di ridurre in nel minor tempo possibile l'impatto sul dispositivo da parte dell'applicazione, quindi ferma le attività che hanno alto impatto sulla cpu, non salva dati, non effettua transazioni su database e non compie a sua volta azioni che possono avere alto impatto sulla cpu.

onRestart() effettua operazioni simili ad onCreate() e viene richiamato soltanto dalle activity che erano state precedentemente fermate.

onStop() è chiamata poco prima che l'activity sia distrutta, esegue tutte le operazioni di terminazione ad alto impatto sulla cpu e salva nel Bundle lo stato di ogni view.

onDestroy() viene invece chiamata subito prima che l'activity venga distrutta (dopo la onStop()), cosa che può succedere perché essa non serve più e viene chiamato il metodo finish(), oppure perché il sistema operativo ha bisogno di spazio nello stack delle activity.

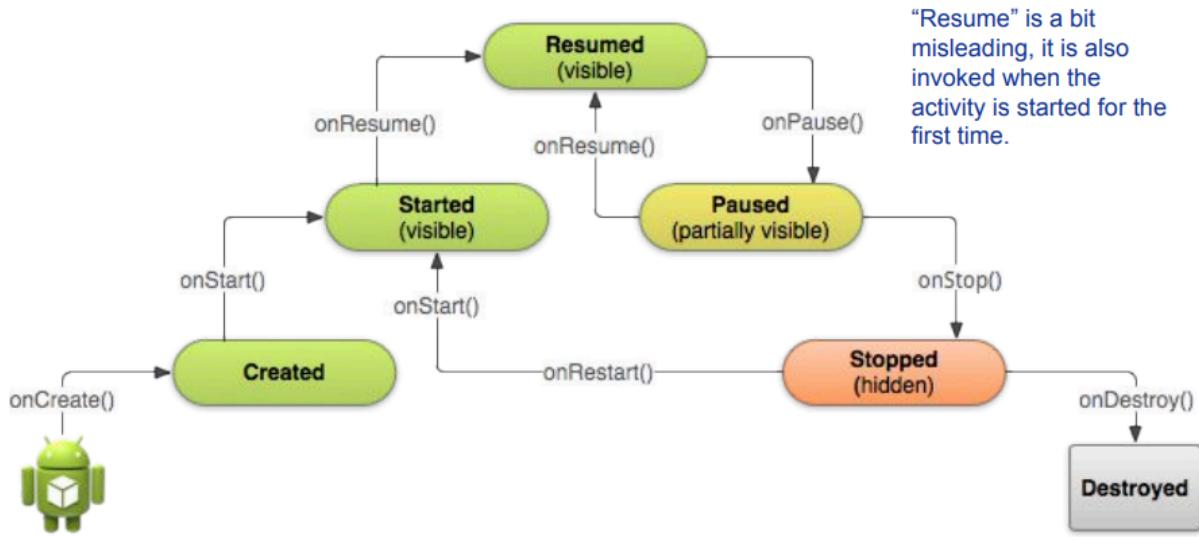
Queste fasi si alternano sostanzialmente in tre cicli principali:



Mainly 3 different loops:

- **Entire lifetime**
 - Between onCreate() and onDestroy().
 - Setup of global state in onCreate()
 - Release remaining resources in onDestroy()
- **Visible lifetime**
 - Between onStart() and onStop().
 - Maintain resources that has to be shown to the user.
- **Foreground lifetime**
 - Between onResume() and onPause().
 - Code should be light.

Di conseguenza gli **stati** in cui si può trovare un'activity sono:



O in altra maniera:

▫ Active (or running)

- Foreground of the screen (top of the stack)

▫ Paused

- Lost focus but still visible
- Can be killed by the system in extreme situations

▫ Stopped

- Completely obscured by another activity
- Killed if memory is needed somewhere else

Ricreare un'activity:

Quando un'activity viene distrutta tutti i dati delle sue view sono automaticamente salvati in un Bundle chiamato **InstanceState** e vengono poi ricaricati quando viene creata una nuova istanza.

Se si vogliono salvare più dati di quelli normalmente conservati nell'Instace State si può:

- sovrascrivere `onSaveInstanceState()` e `onRestoreInstanceState()`
- usare un ViewModel (trattato in seguito)

Esempio:

```

static final String STATE_SCORE = "playerScore";
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    super.onSaveInstanceState(savedInstanceState);
    savedInstanceState.putInt(STATE_SCORE, mCurrentScore);
}

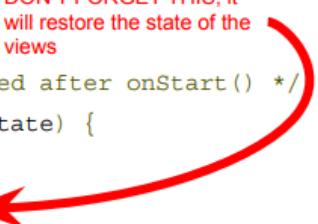
```

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState); // Always call the superclass first
    if (savedInstanceState != null) {
        // Restore value of members from saved state
        mCurrentScore = savedInstanceState.getInt(STATE_SCORE);
    } else {
        // Probably initialize members with default values for a new instance
    }
}
/* The difference is that onRestoreInstanceState is called after onStart() */
public void onRestoreInstanceState(Bundle savedInstanceState) {
    // Call the superclass to restore the views
    super.onRestoreInstanceState(savedInstanceState);
    mCurrentScore = savedInstanceState.getInt(STATE_SCORE);
}

```

DON'T FORGET THIS, it
will restore the state of the
views

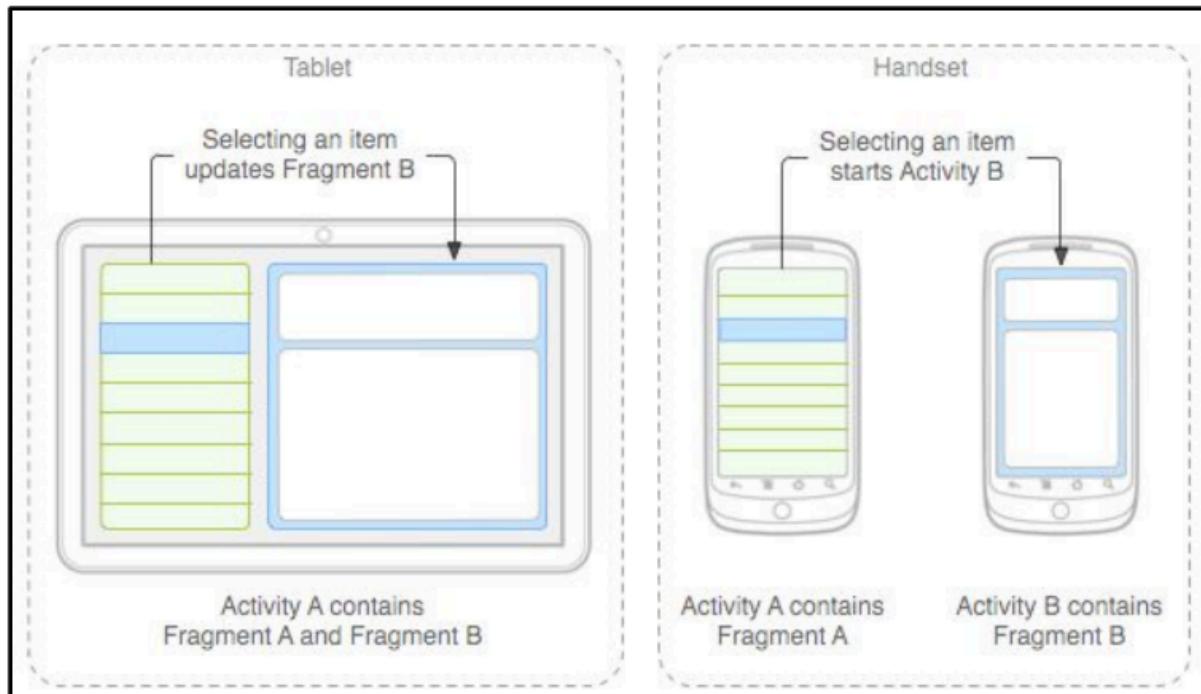


Fragment:

I fragment (o frammenti) sono **porzioni modulari dell'interfaccia utente** di un'activity introdotte con Android 3.

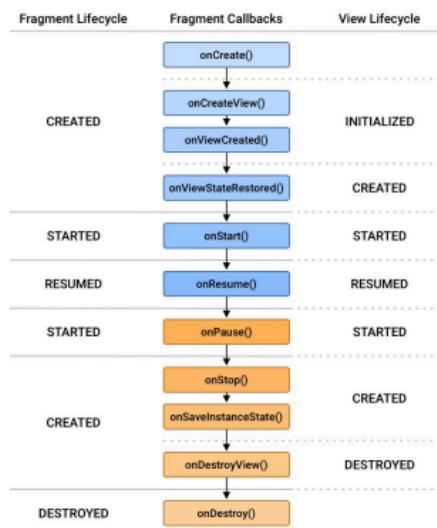
I fragment sono molto utili in quanto possono essere riutilizzati più volte nella stessa activity.

Esempio sull'utilizzo di frammenti:



Ogni fragment ha un suo ciclo di vita direttamente dipendente dall'activity in cui è contenuto, mentre può avere un proprio layout completamente scollegato da quello dell'activity.

i fragment possono essere creati, distrutti e sostituiti a runtime.



The lifecycle of the Activity in which the Fragment lives directly affects the lifecycle of the Fragment.

onPause (Activity) → **onPause** (Fragment)

onStart (Activity) → **onStart** (Fragment)

onDestroy (Activity) → **onDestroy** (Fragment)

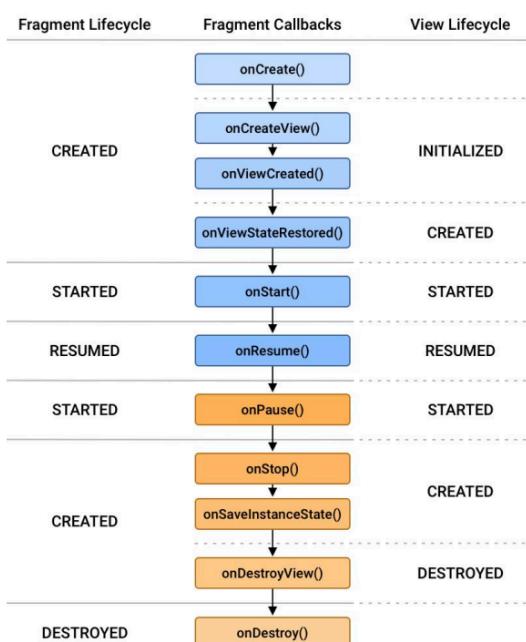
Fragments have also extra lifecycle callbacks to enable runtime creation/destruction.

per istanziare un fragment si estende la classe fragment:

```
public class MyFragment extends Fragment { ...}
```

oppure si possono estendere una serie di fragment già precostituiti come DialogFragment, ListFragment, PreferenceFragmentCompat, ecc...

ciclo di vita di un fragment:



è molto simile a quello di un'activity (perché ne è dipendente).

Si osservino in particolare:

- **onCreate()**: metodo chiamato alla creazione del frammento.
- **onCreateView()**: chiamato quando il frammento viene disegnato per la prima volta o quando viene richiamato dallo stack, ritorna la view associata al frammento.
- **onPause()**: chiamato quando l'utente abbandona il frammento per mettere il focus su qualcosa altro.

Creazione di un fragment:

per creare un fragment si utilizza un oggetto **container** che è il genitore nella view del frammento ed un oggetto **inflater** che serve per popolare il fragment con un layout che gli viene fornito come parametro.

L'inflater ritorna la gerarchia di view che ha istanziato.

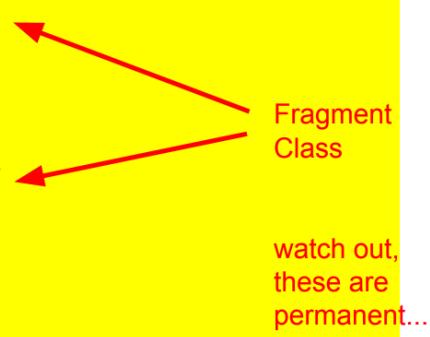
```
public class ExampleFragment extends Fragment {  
  
    @Override  
    public View onCreateView(LayoutInflater inflater,  
        ViewGroup container, Bundle savedInstanceState) {  
        return inflater.inflate(R.layout.example_fragment,  
            container, false);  
    }  
}
```

Questo però è un modo vecchio di farlo, adesso il costruttore della classe fa esattamente la stessa cosa e di conseguenza si tende ad usare quello.

```
public class ExampleFragment extends Fragment {  
  
    public ExampleFragment () {  
        super(R.layout.example_fragment);  
    }  
}
```

Per aggiungere un fragment alla UI dal file XML lo si definisce come una view:

```
<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="fill_parent"  
    android:layout_height="fill_parent"  
    android:orientation="horizontal" >  
    <fragment android:name="it.cs.android30.FragmentOne"  
        android:id="@+id/f1"  
        android:layout_width="wrap_content"  
        android:layout_height="fill_parent"  
        />  
    <fragment android:name="it.cs.android30.FragmentTwo"  
        android:id="@+id/f2"  
        android:layout_width="wrap_content"  
        android:layout_height="fill_parent"  
        />  
</LinearLayout>
```



Attenzione: i fragment definiti nel file xml non sono eliminabili o sostituibili a runtime, però si possono utilizzare come modelli per fare linflate di altri frammenti definiti programmaticamente.

Quindi al posto di mettere nell'XML direttamente i fragment conviene mettere una FragmentContainerView, contenente a sua volta i fragment che si vogliono inserire e che sono sostituibili a runtime.

```

<fragment android:name="it.cs.android30.FragmentOne"
    android:id="@+id/f1"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
/>

<androidx.fragment.app.FragmentContainerView android:name="it.cs.android30.FragmentOne"
    android:id="@+id/f1"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
/>

```

i frammenti in fragmentContainerView devono necessariamente trovarsi in un layout.

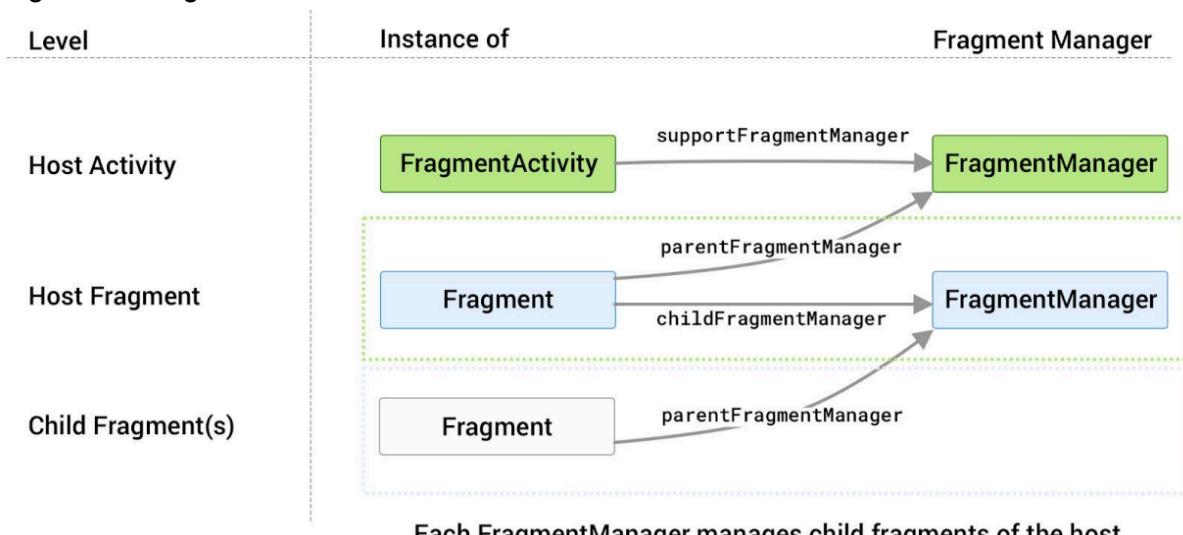
In particolare quando un fragment viene istanziato da un file XML il sistema compie le seguenti azioni:

- assegna il layout all'activity.
- crea il frammento istanziando la corrispettiva classe e chiamando onCreate().
- chiama onCreateView() e tramite l'inflater inserisce le informazioni relative al suo contenuto ed alla sua posizione.

FragmentManager:

Un'activity per interagire con i suoi fragment utilizza il suo fragmentManager.

Anche un fragment può contenere altri fragment, in questo caso ha anche lui un suo fragmentManager.



Per ottenere il fragmentManager avendo un'activity si chiama **getSupportFragmentManager()**, mentre per ottenerlo partendo da un frammento si usa **getParentFragmentManager()**.

Un fragment può ritornare la sua activity tramite **getActivity()**, ma chiamare funzioni su questa activity è complesso e pericoloso, infatti un fragment può essere usato da più activity e di conseguenza non si è mai certi del tipo di activity che lo sta contenendo in quel momento.

Invece un'activity può ottenere un riferimento ai suoi frammenti nella seguente maniera:

```
ExampleFragment fragment=(ExampleFragment)
getSupportFragmentManager().findFragmentById(R.id.example_fragment)
```

Alla creazione un frammento chiama **onAttach()** quando viene passato al fragmentManager ed **onDetach()** quando viene tolto dal fragmentManager prima di essere distrutto.

Di conseguenza se un frammento deve notificare la sua presenza all'activity lo deve fare in **onAttach()**.

Esempio:

```
public static class FragmentA extends ListFragment {
    OnArticleSelectedListener listener;
    ...
    @Override
    public void onAttach(Context context) {
        super.onAttach(context);
        try {
            listener = (OnArticleSelectedListener) context;
        } catch (ClassCastException e) {
            throw new ClassCastException(context.toString() + " must implement OnArticleSelectedListener");
        }
    }
}
```

Fragment transactions:

l'attività di aggiungere, rimuovere o sostituire un frammento è detta Transaction e viene salvata in una **pila contenuta nell'activity**.

Però mentre la gestione della pila delle activity viene fatta automaticamente, per i fragment va fatta manualmente, quindi per fare una transazione si fa una cosa del tipo:

1. ACQUIRE an instance of the FRAGMENT MANAGER

```
FragmentManager fm = getSupportFragmentManager();
FragmentTransaction transaction = fm.beginTransaction();
transaction.setReorderingAllowed(true);
```

2. CREATE new Fragment and Transaction (changes you want at the same time)

```
FragmentExample newFragment = new FragmentExample();
transaction.replace(R.id.fragment_container, newFragment);
```

3. SAVE to backStack and COMMIT

```
transaction.addToBackStack("FragmentExample");
transaction.commit();
```

FragmentActivity then automatically retrieves fragments from the back stack via **onBackPressed()** una transazione non viene applicata finché non viene fatto il **commit**, a quel punto se è stato invocato **addToBackStack()** allora il frammento viene stoppato, messo nello stack ed è possibile recuperarlo facendo **popBackStack()** (il pulsante indietro di android fa questa azione di default), se invece non è stata chiamata il frammento viene distrutto e non sarà più possibile recuperarlo.

Intents:

un intent è un messaggio utile al collegamento **late runtime** tra componenti di diverse applicazioni.

Per late runtime si intende che il collegamento è fatto soltanto nel momento in cui è fatta la chiamata alla funzione dell'intent, non quando è avviata l'applicazione (early runtime) o quando l'applicazione è compilata (compile time binding).

Le componenti (components) che possono ricevere intent sono:

- **Activities**
- **Services** (activity senza interfaccia grafica)
- **BroadcastReceivers**

In un'applicazione per passare da un'activity ad un'altra si manda un intent.

Gli intenti possono essere esplicativi o impliciti:

- **Esplicito**: è utilizzato per lanciare delle activity specifiche, il destinatario è specificato in Component Name;
- **Implicito**: il tipo di destinatario è specificato in Data o in action Name ed è il sistema operativo a scegliere quale activity è il destinatario (le activity hanno il loro tipo e cosa possono fare nel loro manifest).

Struttura di un intent:



Component Name: indica il componente che dovrebbe ricevere l'intent, è opzionale e va specificato solo nel caso l'intent sia esplicito. Il valore si inserisce con void setComponent(ComponentName)

Action Name: una stringa contenente il nome dell'azione che deve essere eseguita.

Queste azioni sono predefinite o possono essere create dal programmatore, si inseriscono con void setAction(String).

Azioni predefinite:

Action Name	Description
ACTION_EDIT	Display data to edit
ACTION_MAIN	Start as a main entry point, does not expect to receive data.
ACTION_PICK	Pick an item from the data, returning what was selected.
ACTION_VIEW	Display the data to the user
ACTION_SEARCH	Perform a search
ACTION_SEND	Send some data through another component

ACTION_IMAGE_CAPTION	Open the camera and receive a photo
ACTION_VIDEO_CAPTION	Open the camera and receive a video
ACTION_DIAL	Open the phone app and dial a phone number
ACTION_SENDTO	Send an email (email data contained in the extra)
ACTION_SETTINGS	Open the system setting
ACTION_WIRELESS_SETTINGS	Open the system setting of the wireless interfaces
ACTION_DISPLAY_SETTINGS	Open the system setting of the display

Azioni definite dal programmatore:

`it.example.projectpackage.FILL_DATA` (package prefix + name action)

Data: è un URI (detto **Name**) che fa riferimento ad un qualche file ed il suo tipo mime (detto **Type**).

Name e type si impostano con void setData(Uri) e void setType(String), ma questi due metodi mettono a null l'altro campo quando vengono invocati, di conseguenza per inserire entrambi i valori si utilizza setDataAndType().

Un name che inizia con content indica un file presente sul dispositivo.

Esempi:

▫ **name:** Uniform Resource Identifier ([URI](#))

▫ **type:** **MIME** (Multipurpose Internet Mail Extensions)-type

▫ Composed by two parts: a [type](#) and a [subtype](#)

▫ **scheme://host:port/path**

[EXAMPLEs](#)

`tel://003-232-234-678`

`content://contacts/people`

`http://www.cs.unibo.it/`

Image/gif image/jpeg image/png image/tiff
 text/html text/plain text/javascript text/css
 video/mp4 video/mpeg4 video/quicktime video/ogg
 application/vnd.google-earth.kml+xml

Category: campo che contiene informazioni sul tipo di component che deve gestire l'intent. Contiene normalmente un valore di default, ma può essere modificato con void addCategory(String).

Tipi di category:

Category Name	Description
CATEGORY_HOME	The activity displays the HOME screen.
CATEGORY_LAUNCHER	The activity is listed in the top-level application launcher, and can be displayed.
CATEGORY_PREFERENCE	The activity is a preference panel.
CATEGORY_BROWSABLE	The activity can be invoked by the browser to display data referenced by a link.

Extra: è un insieme di coppie chiave-valore che può contenere dati vari necessari al gestore dell'intent, tali coppie si inseriscono con void putExtras() e si ottengono con getExtras().

Gli extra hanno tipi predefiniti, ma lo sviluppatore può definire i propri con la sintassi:

```
static final String EXTRA_BASS = "it.example.BASS_NOTE";
```

Flags: informazioni addizionali di tipo booleano che servono al sistema per trattare nella maniera appropriata l'intent.

Intent esplicativi:

Per istanziare un intent esplicito si inserisce l'activity destinatario nel costruttore e lo si spedisce con startActivity(intent):

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

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

Intent impliciti:

Negli intent impliciti non è specificata l'activity destinatario (il component name è lasciato vuoto), ma sono indicati il tipo di intent tramite la sezione Data (definendo i campi type e name) e il tipo di azione che chi riceve l'intent deve compiere tramite action name (essenziale).

Si può creare un'action name personalizzato oltre a quelli predefiniti.

chi deve ricevere gli intent deve indicare nel suo **intent filter** nel manifest che intent può accettare:

```
<intent-filter>
    <action android:name="my.project.ACTION_ECHO" />
</intent-filter>
```

Il sistema operativo fa tre test sull'intent filter:

- **action test:** l'azione dell'intent deve corrispondere ad una specificata nell'intent filter, se l'intent non specifica nessuna azione va bene per tutte le azioni dell'intent filter (che deve averne almeno una);
- **category test:** ogni categoria dell'intent deve corrispondere ad una nell'intent filter, se nell'intent non è specificata alcuna categoria Android assume sia CATEGORY_DEFAULT (che quindi deve essere presente nell'intent filter);
- **data test:** vengono fatti dei test sullo schema e sul tipo di dato verificando che siano compatibili con quanto specificato nell'intent filter.

Se vengono passati tutti i tre test l'applicazione viene inserita tra quelle che possono rispondere all'intent, se c'è n'è più di una viene chiesto all'utente quale utilizzare.

Intent con risposta:

Esistono inoltre intent che possono ricevere messaggi in risposta, in questo caso si usa (sintassi deprecata ma noi la vediamo lo stesso, nella realtà si deve usare Activity Result API che è trattata più in basso) `startActivityForResult(Intent intent, int requestCode)` e `onActivityResult(int requestCode, int resultCode, Intent data)`, dove requestCode è un codice identificativo di quella comunicazione:

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

...
public void onActivityResult(int requestCode, int resultCode, Intent data)
{
    // Invoked when SecondActivity completes its operations ...
}
```

Mentre chi risponde deve fare:

```
Intent intent = getIntent();
setResult(RESULT_OK, intent);
intent.putExtra("result", resultValue);
finish();
```

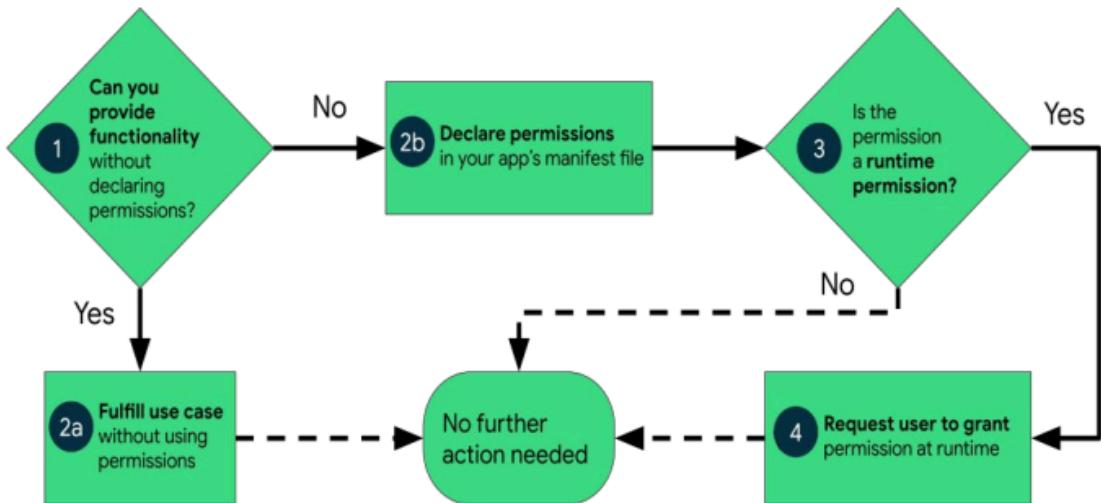
Activity Result API:

Utilizzare il metodo descritto sopra per lanciare e raccogliere gli intent è sconsigliato, infatti nel caso l'activity mittente sia distrutta e ricreata mentre il destinatario si sta eseguendo si hanno delle pessime conseguenze.

Per superare questo problema si deve utilizzare l'Activity Result API.

Permessi di Android:

Attualmente il criterio con cui si devono richiedere i permessi è il seguente:



Ma nel tempo questa cosa è cambiata molto:

Fino ad android 6.0 (escluso) era sufficiente dichiarare nel manifest cosa si sarebbe utilizzato.

Da android 6.0 in poi invece la concessione dei permessi viene gestita in modo da essere maggiormente sotto il controllo dell'utente: esso può concedere solo alcuni permessi, rifiutarne altri e cambiare queste impostazioni in un qualunque momento.

Per questo motivo quando si utilizza una risorsa è necessario verificare sempre di avere il relativo permesso (se si prova ad utilizzare una risorsa su cui non si hanno i permessi il programma crasha).

Check if permission is granted

When: before performing a permission needed action

```

if (ContextCompat.checkSelfPermission(thisActivity, Manifest.permission.ACCESS_FINE_LOCATION)
!= PackageManager.PERMISSION_GRANTED) {
    // Permission is not granted
}
  
```

If it is not requested, ask for it

```

ActivityCompat.requestPermissions(thisActivity,
    new String[]{Manifest.permission.ACCESS_FINE_LOCATION},
    MY_PERMISSIONS_LOCATION);
  
```

Wait for the user response asynchronously

When: before performing a permission needed action

```
@Override  
public void onRequestPermissionsResult(int requestCode, String permissions[], int[] grantResults) {  
    switch (requestCode) {  
        case MY_PERMISSIONS_LOCATION: {  
            if (grantResults.length > 0  
                && grantResults[0] == PackageManager.PERMISSION_GRANTED) {  
                // GRANTED  
            } else { // DENIED  
                return;  
            }  
        }  
    }  
}
```

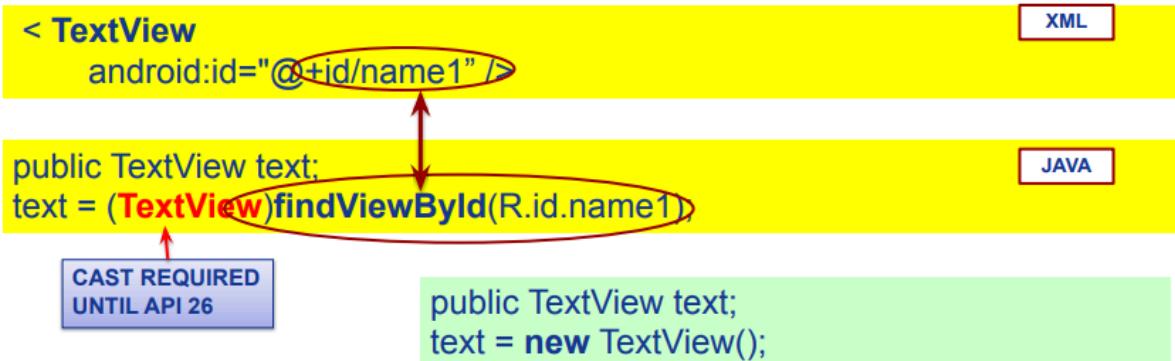
View:

Una view è un'area rettangolare responsabile del **disegno** del proprio contenuto e della gestione di **eventi**.

Un widget è una sottostruttura di una view.

Ogni view ha un focus (se l'utente è attivo su quella view o su un'altra, si può forzare tramite requestFocus()) ed una visibilità (se l'utente può vedere o meno la view, si può impostare tramite setVisibility(int)).

Le view possono essere create tramite l'XML, via codice in java oppure create nell'XML e poi accedute da codice in un secondo momento.



Eventi:

Quando l'utente interagisce con le view vengono generati degli eventi che possono essere catturati e gestiti con i seguenti metodi:

- Tramite funzioni di **callback** indicate direttamente nel file **XML**:

```
<Button
    android:text="@string/textButton"
    android:id="@+id/idButton"
    android:onClick="doSomething"
/>>
```

XML Layout File

Java class

```
public void doSomething(View w) {
    // Code to manage the click event
}
```

dove doSomething è una funzione dell'activity; il problema sorge quando si riutilizza lo stesso layout da un'altra parte dove doSomething non è definito.

- **Event handler:** viene estesa la classe View sovrascrivendo le funzioni di callback standard, per esempio estendendo la classe Button si sovrascrive il metodo onTouchEvent() assegnandogli il comportamento voluto alla pressione del pulsante.
- **Event listener:** interfacce esterne (single abstract method interface) fatte in modo che chi le estende implementi funzioni di gestione degli eventi.

Button

interface OnClickListener

```
public abstract void onClick(View v) {}
```

```
public class ExampleActivity extends Activity implements OnClickListener {
    ...
    Button button = findViewById(R.id.buttonNext);
    button.setOnClickListener(this);
    ...
    public void onClick(View v) { <behavior...> }
}
```

Layout:

i layout sono delle view.

una sottoclasse della view è una viewgroup, ovvero una view che può contenere altre view, un layout è una sottoclasse di viewgroup.

un layout è una viewgroup che ha dei parametri in più (come layout_height o layout_width).

NOTA: gli elementi a schermo vengono disegnati nell'ordine in cui vengono letti dall'alto al basso nel file xml.,

Esistono diverse sottoclassi di layout:

LinearLayout: dispone tutte le view su una singola riga o colonna

RelativeLayout: poco utilizzata negli ultimi anni, si definisce un elemento e poi tutti gli altri in relazione ad esso.

TableLayout: dispone le view in una tabella ed anche esso è utilizzato di rado.

FrameLayout: contiene uno solo elemento che riempie tutta l'area.

ConstraintLayout: è il layout di default ed il più utilizzato.

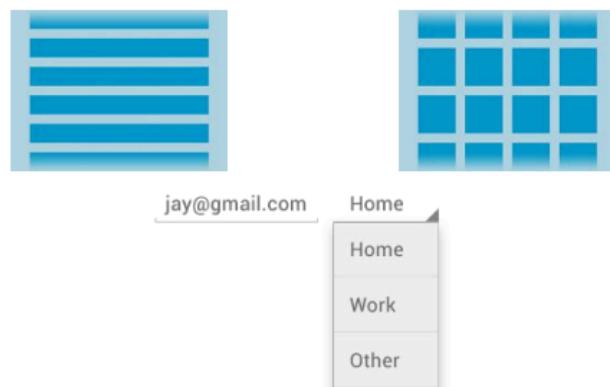
Ogni elemento deve avere almeno due vincoli (uno orizzontale ed uno verticale), ovvero delle relazioni con degli altri elementi del layout che ne definiscono la posizione.

Ha dei widget in più chiamate guideLine e che sono delle linee alle quali le altre view possono fare riferimento.

Dynamic layout: ci sono tipi di visualizzazioni che si devono adattare al dato che esse contengono, per fare questo si utilizzando delle sottoclassi di AdapterView (a sua volta sottoclasse di View).

Esempio:

- ❖ Some subclasses:
 - ❖ ListView
 - ❖ GridView
 - ❖ Spinner
 - ❖ Gallery



per utilizzare un adapter si fa:

```
ArrayAdapter<String> adapter = new
    ArrayAdapter<String>(this,
        android.R.layout.simple_list_item_1,
        myStringArray);
ListView listView = (ListView)
    findViewById(R.id.listview);
listView.setAdapter(adapter);
```

Al posto di usare gli adapter da soli adesso si utilizzano le recyclerView che gestiscono i dati meglio.

Per ogni recyclerView si deve definire un viewHolder che è una struttura che contiene i riferimenti a tutti gli elementi che devono essere visualizzati nella view.

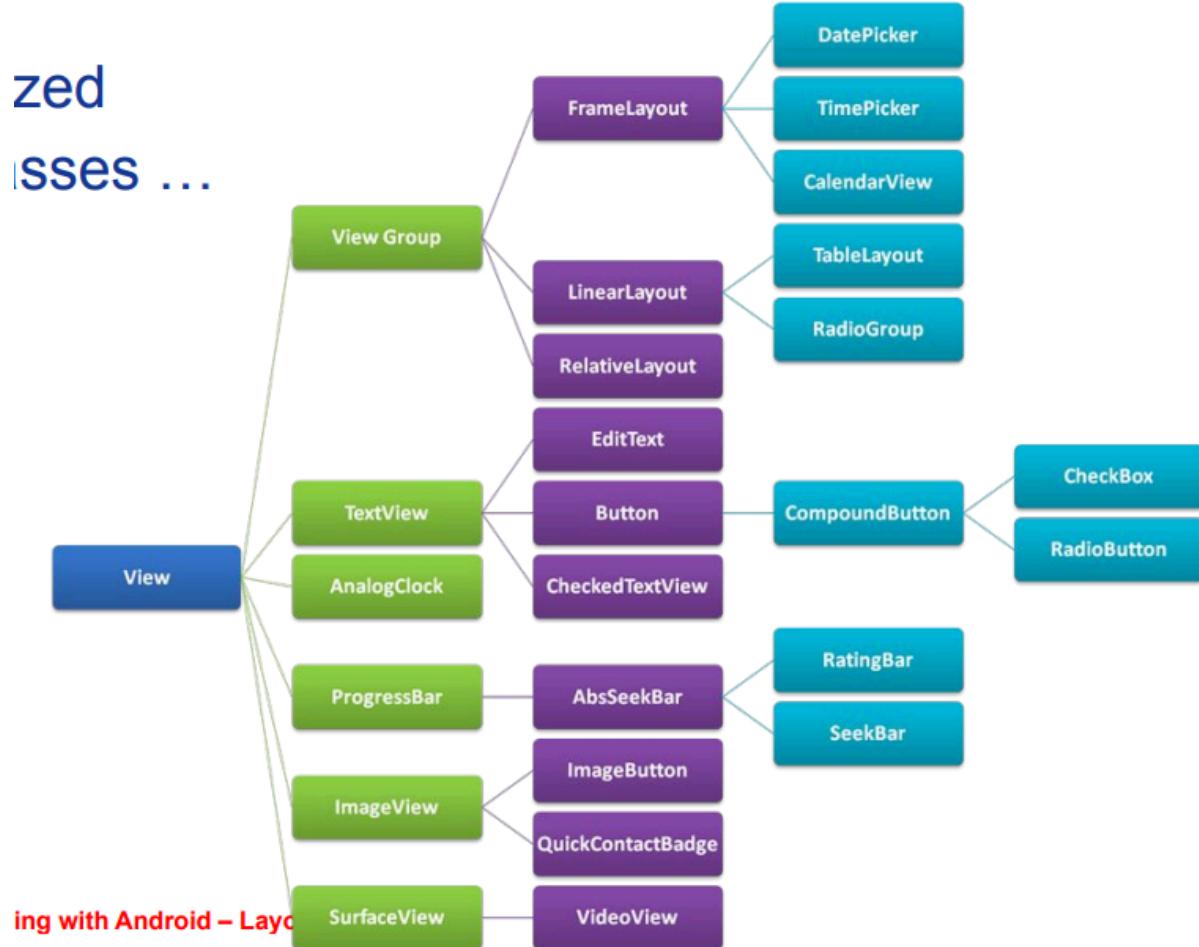
Quando si implementa questo adapter di devono sovrascrivere i seguenti metodi:

- getItemCount()
- onCreateViewHolder()
- onBindViewHolder(): qui si definisce il comportamento di ogni elemento della view.

CardView:

sono dei layout fatti apposta per essere utilizzati in una recyclerView e rappresentano un elemento di una lista.

Gerarchia delle view:



TextView:

view che può contenere stringhe o html, è generalmente utilizzata per contenere informazioni statiche.

può contenere anche dei link (con una start activity inserita automaticamente).

EditView:

view simile alla textView ma in cui l'utente può inserire dei dati.

i suggerimenti per l'autocompletamento vengono realizzati tramite un adapter.

```
String[] tips = getResources().getStringArray(R.array.nani_array);
ArrayAdapter<String> adapter = new ArrayAdapter(this,
        android.R.layout.simple_dropdown_item_1line, tips);
AutoCompleteTextView acTextView=(AutoCompleteTextView)
        findViewById(R.id.inputText);
acTextView.setAdapter(adapter);
```

Button:

sono i pulsanti.

copia tutte le sottoclassi

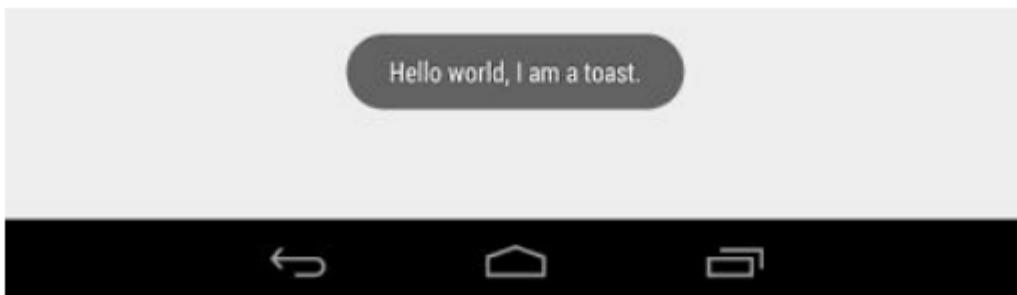
Spinner:

sono i menù a tendina.

anche in questo caso le opzioni del menù a tendina vengono realizzate tramite un adapter.

Toast:

piccoli messaggi associati ad un'activity simili agli alert utilizzati per mostrare informazioni all'utente.



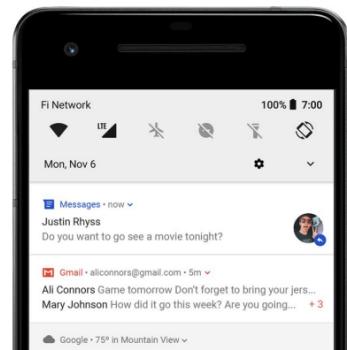
Notifiche:

sono messaggi mandati da un'applicazione ad una pool universale di notifiche gestite dal sistema operativo.

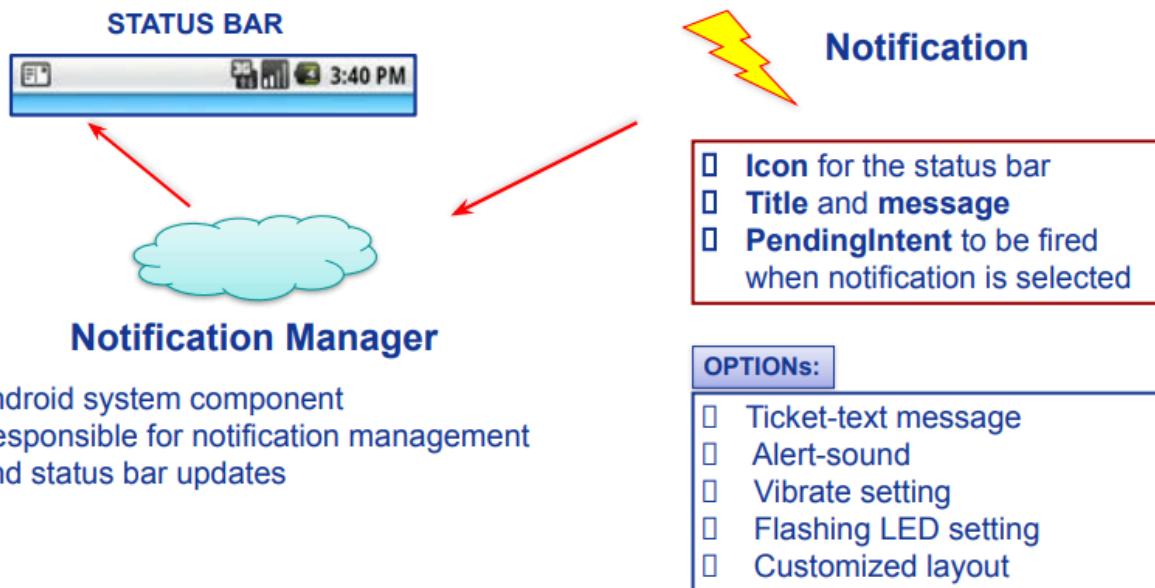
Questi messaggi possono essere unicamente informativi o anche interattivi, dove l'utente può interagire con la notifica stessa o con dei suoi pulsanti.

Il sistema potrebbe anche ignorare le notifiche e non mostrarle all'utente (per vari motivi, come per esempio aver finito la memoria).

Lo sviluppatore può personalizzare la quantità di informazioni mostrate nelle notifiche e come l'utente ci può interagire.

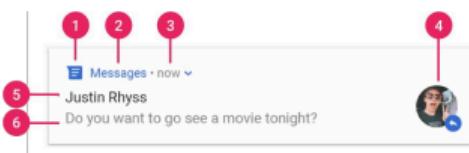


Da codice si utilizza un notification Builder per creare la notifica e poi la si manda al notification manager del sistema operativo.

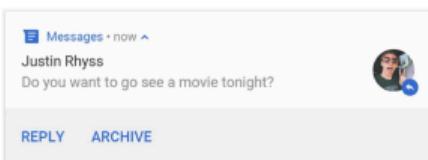


PendingIntent è un intent che viene mandato nel momento in cui l'utente preme sulla notifica o su uno dei suoi pulsanti.

Struttura di una notifica:



1. Small icon
2. App name
3. Timestamp
4. Optional Large Icon
5. Optional Title
6. Optional Text



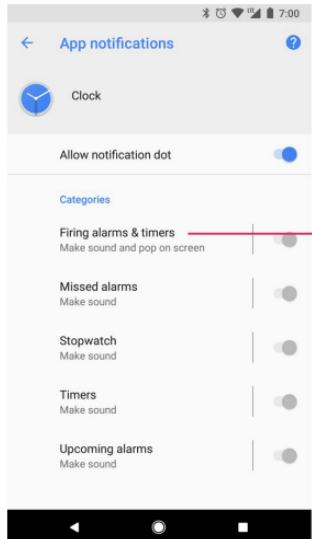
Starting with Android 7.0, users can perform simple actions directly in the Notification

Le notifiche possono inoltre anche essere **aggiornate**: quando si manda la notifica al sistema operativo vi si associa un **id** che può essere utilizzato per mandare una nuova notifica che rimpiazza la precedente.

Da Android 7 è anche possibile raggruppare le notifiche in gruppi.

Da android 8 le notifiche devono essere associate ad un canale che indichi di che categoria fanno parte, inoltre i vari canali hanno priorità diverse.

Esempio di canali:



NotificationCompat:

In questo corso si utilizza per fare le notifiche il modulo NotificationCompat.

Per mandare una notifica si utilizza il design pattern builder

(<https://refactoring.guru/design-patterns/builder>):

1. Get a reference to the Notification Manager

```
NotificationManager nm = (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE)
or (better)
```

```
NotificationManagerCompat nm = NotificationManagerCompat.from(this);
```

2. Build the Notification message (design pattern Builder)

```
NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this, CHANNEL_ID);
mBuilder.setContentTitle("Picture Download").setContentText("Download in progress")
.setSmallIcon(R.mipmap.ic_launcher_round).setPriority(NotificationCompat.PRIORITY_DEFAULT);
```

3. Send the notification to the Notification Manager

```
notificationManager.notify(myId, mBuilder.build());
```

Per inserire un PendingIntent nella fase 2 di costruzione si fa:

```
Intent newIntent = new Intent(this, ReceivingActivity.class);
newIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
    Intent.FLAG_ACTIVITY_CLEAR_TASK);
newIntent.putExtra("CALLER", "notifyService");
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, newIntent, PendingIntent.FLAG_IMMUTABLE);

mBuilder.setContentIntent(pendingIntent);
```

A Class in your APP, such as a normal intent

Mentre per aggiungerlo sotto forma di pulsante:

```
mBuilder.addAction(R.drawable.ic_notification, "PRESS ME", pendingIntent);
```

Opzioni per personalizzare le notifiche:

Add (optional) flags for notification handling

```
mBuilder.setAutoCancel(true)
```

□ Notification goes away when tapped

Send the notification to the Notification Manager

```
notificationManager.notify(0, mBuilder.build());
```

Add a **long text** and make the notification expandable

```
mBuilder.setStyle(new NotificationCompat.BigTextStyle()
    .bigText("Much longer text that cannot fit one line..."))
```

Add a **sound** to the notification

```
mBuilder.setSound(URI sound);
```

Add **flashing lights** to the notification

```
mBuilder.setLights(0xff00ff00, 300, 100);
```

This sets a green led

The LED flashes for 300ms and turns it off for 100ms

Add a **vibration pattern** to the notification

```
mBuilder.setVibrate(long [])
mBuilder.setVibrationPattern(long []) // From API 26
```

BoilerPlate per la creazione di un canale di notifica:

Set Notification Channels from Android 8.0 (API 26)

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    CharSequence name = getString(R.string.channel_name);
    String description = getString(R.string.channel_description);
    int importance = NotificationManager.IMPORTANCE_DEFAULT;
    NotificationChannel channel =
        new NotificationChannel(CHANNEL_ID, name, importance);
    channel.setDescription(description);
    NotificationManager notificationManager =
        getSystemService(NotificationManager.class);
    notificationManager.createNotificationChannel(channel);
}
```

Operazioni in background:

è buona norma mettere in un thread in background ogni operazione che può potenzialmente impiegare più di qualche millisecondo per eseguirsi (per esempio la creazione di notifiche). Infatti se l'interfaccia utente si blocca per più di 5 secondi (e generalmente l'interfaccia si trova sul main thread) il sistema inizia a notificare l'utente che il programma non risponde. Ogni operazione viene eseguita di default sul thread principale (main thread), ma è possibile indicare nel manifest quali componenti eseguire su thread indipendenti oppure creare thread a runtime da codice.

I processi possono essere uccisi dal sistema operativo nel caso ci sia bisogno di memoria e vengono scelti secondo questa gerarchia:

Foreground, Visible, Service, Background, Empty.

Per creare un thread si può:

- estendere la classe Thread
- implementare l'interfaccia Runnable
- usare AsyncTask, ma questo metodo è deprecato
- utilizzare le Coroutines (soltanto in Kotlin)

In questo corso vediamo in particolare le prime due modalità.

Estendere la classe Thread:

si crea una nuova classe figlia di **Thread** contenente nel metodo **run()** il codice che si vuole eseguire, la si istanzia e si chiama il metodo **start()** che ne fa partire l'esecuzione.

```
public class MyThread extends Thread {  
  
    public MyThread() {  
        super ("My Thread");  
    }  
  
    public void run() {  
        // do your stuff  
    }  
}
```

e poi:

```
myThread m = new MyThread();  
m.start();
```

Implementare l'interfaccia Runnable:

Per utilizzare questo metodo si deve prima creare una **Thread pool**.

Una thread pool è un insieme preallocato di thread sui quali vengono eseguiti degli oggetti che implementano l'interfaccia Runnable.

Questi oggetti vengono prelevati da una coda man mano che i thread delle thread pool si svuotano.

Con questa tecnica si evita di creare programmi che generano un numero spropositato di thread.

Per creare una thread pool si crea un oggetto di tipo ExecutorService:

```
ExecutorService executorService = Executors.newFixedThreadPool(4);
```

Poi si aggiunge alla sua coda un oggetto che implementa l'interfaccia Runnable e che contiene il codice che si vuole eseguire in maniera asincrona:

```
executorService.execute(new Runnable() {
    @Override
    public void run() {
        // do your stuff
    }
});      // See also Lambda notation
```

Message passing:

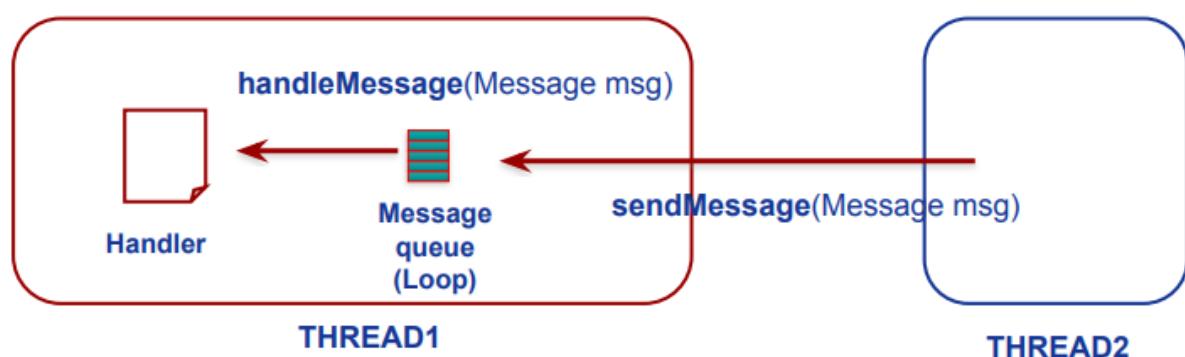
Utilizzare un thread che non è il main per disegnare a schermo si può fare ma può generare race condition, quindi è buona norma che questa cosa la faccia solo il main thread.

Di conseguenza è necessario che i thread comunichino tra di loro, per fare ciò si utilizzano:

- handlers e loopers
- AsyncTask (soluzione storica deprecata)
- Observables

Handlers e loopers:

Il processo che vuole ricevere messaggi implementa un **Handler** che ha una funzione di callback per reagire ai **messaggi** che si ricevono, tale handler fa parte dell'oggetto **Looper** che è colui il quale controlla l'arrivo di nuovi messaggi e li conserva in una **coda**.



Per fare ciò quando si implementa un nuovo processo si fa:

```

public void run() {
    Looper.prepare(); // Instantiate the queue
    handler = new Handler(Looper.myLooper()) {
        @Override
        public void handleMessage(Message msg) {
            // handle here the message
        }
    }
    Looper.loop(); // Have it ready for receiving
}

```

Oppure al posto di Thread si estende **HandlerThread** che ha già un Looper di default.

Invece per **mandare** un messaggio ad un altro thread:

HOW? Let's imagine the thread of the previous slide is called mThread

```

mThread.start();
Handler mHandler = mThread().handler; // Assuming you can get the handler

```

You can send it a message to be handled by the **handleMessage**

```

Message m = mThread.handler.obtainMessage(); // new message for mHandler
m.arg1 = "Argument for the message";
mThread.handler.sendMessage(m);

```

OR something to execute on the thread that owns the handler

```

mThread.handler.post(new Runnable() {
    @Override
    public void run() { /* Something to do */ }
});

```

AsyncTask:

Classe deprecata fatto per avere dei thread in background con output sull'UI thread.

AsyncTask is a Thread helper class (Android only).

- ❖ Computation running on a **background** thread.
- ❖ Results are published on the **UI** thread.
- ❖ Should be used for short operations

RULES

- AsyncTask must be created on the UI thread.
- AsyncTask can be executed only once.
- AsyncTask must be canceled to stop the execution.

```
private class MyTask extends AsyncTask<Par, Prog, Res>
```

Must be subclassed to be used

Par → type of parameters sent to the AsyncTask

Prog → type of progress units published during the execution

Res → type of result of the computation

EXAMPLES

```
private class MyAsyncTask extends AsyncTask<Void,Void,Void>
```

```
private class MyAsyncTask extends AsyncTask<Integer,Void,Integer>
```

EXECUTION of the ASYNCTASK

The UI Thread invokes the **execute** method of the AsyncTask:

```
(new MyAsyncTask()).execute(param1, param2 ... paramN)
```

After **execute** is invoked, the task goes through four steps:

1. **onPreExecute()** □ invoked on the UI thread
2. **doInBackground(Params...)** □ computation of the AsyncTask
 - ❖ can invoke the publishProgress(Progress...) method
3. **onProgressUpdate(Progress ...)** □ invoked on the UI thread
4. **onPostExecute(Result)** □ invoked on the UI thread

Services:

Sono componenti fatti per eseguire **operazioni in background**.

Di default si trovano anche essi sul **main thread**, di conseguenza se le operazioni che devono eseguire sono lunghe è necessario spostarle in ogni caso su un altro thread (o bloccheranno altre activity o service che si stanno eseguendo sullo stesso thread).

I service sono sostanzialmente l'unico modo di fare cose in background quando non ci sono activity attive, infatti se si fanno eseguire a delle activity delle operazioni su altri thread esse verranno fermate quando l'activity perderà visibilità.

Schematicamente:

- **Activity** → UI, can be disposed when it loses visibility
- **Service** → No UI, disposed when it terminates or when it is terminated by other components

I service hanno una priorità inferiore alle activity ed in caso di emergenza vengono terminati per primi.

Per inserire un service nel manifest si fa:

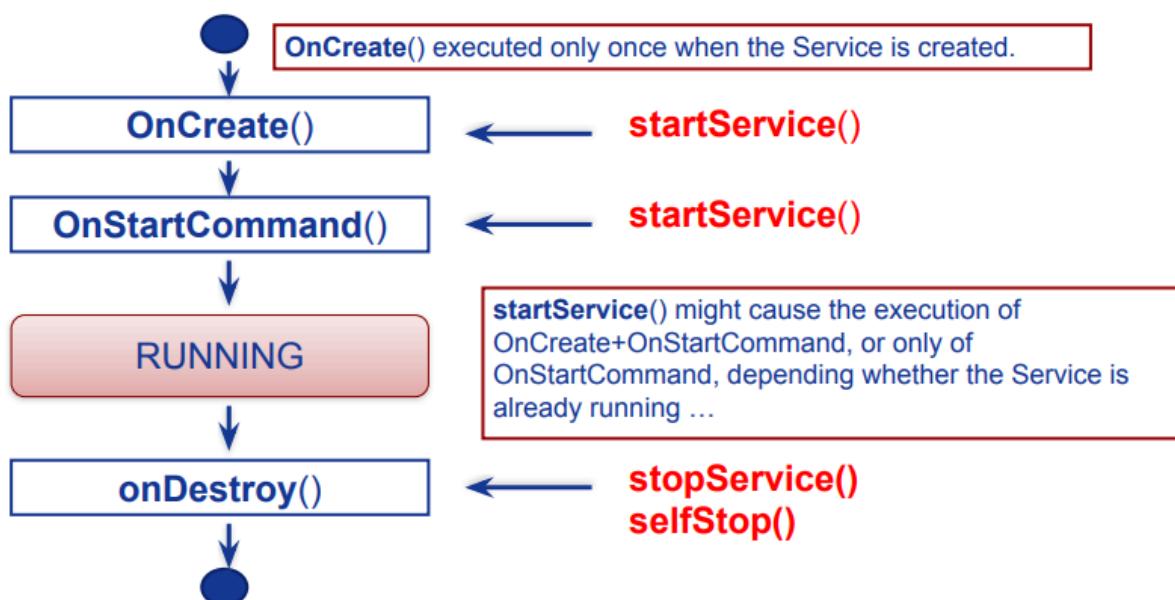
```
<service android:name=".ExampleService" />
```

Oppure per crearne uno da codice si chiama **startService(Intent)** (per interagire con i service è buona norma utilizzare intent espliciti).

Invece la terminazione di un service può avvenire in una delle seguenti maniere:

- uccisione da parte del sistema (per esempio in una situazione di emergenza in cui è necessario liberare memoria),
- il service chiama **stopSelf()**,
- un'altra entità utilizza **stopService(Intent)**.

Ciclo di vita di un service:



Esistono alcuni tipi di service già precostituiti che si possono estendere ed utilizzare, per esempio IntentService.

IntentService:

è un service che presenta una sola funzione di callback che viene chiamata quando arriva un intent e poi si distrugge.

Quello che fa nella funzione di callback viene eseguito in un nuovo thread.

```
public class myIntentService extends IntentService {  
    public myIntentService() { super(" myIntentService"); }  
    @Override  
    protected void onHandleIntent(Intent intent) { // doSomething }  
}
```

ForegroundService:

è un service che è costantemente attivo nella barra di notifica (ha una notifica perennemente visibile nella status bar) ed ha una priorità più alta rispetto agli altri service.

To create a Foreground Service:

- Create a **Notification** object
- Call **startForeground(id, notification)** from **onStartCommand()**
- Call **stopForeground()** to bring it to the background.

Inoltre per creare un foreground service è necessario avere il permesso FOREGROUND_SERVICE.

BoundService:

è un service che dipende da un altro component ed il cui ciclo di vita è strettamente legato a quel component.

Tramite l'oggetto IBinder si controlla l'interazione tra service ed oggetto a cui è bindato.

Broadcast Receiver:

è un componente fatto per ricevere intent.

Quando riceve un intent fa succedere qualcosa indipendentemente dall'activity attiva.

Si può inserire sia nel manifest o nel codice java, la differenza è che nel codice java non può partire mentre l'app è chiusa perchè quel pezzo di codice non viene eseguito.

Nel manifest:

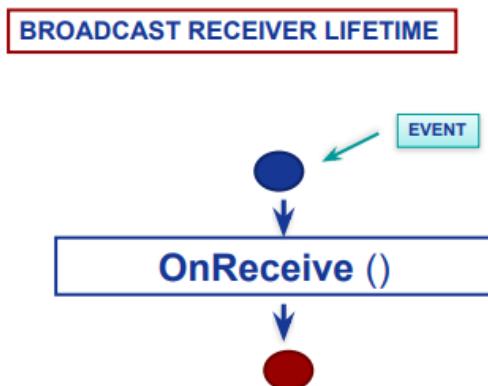
```
<application>
    <receiver class="SMSReceiver">
        <intent-filter>
            <action android:value="android.provider.Telephony.SMS_RECEIVED" />
        </intent-filter>
    </receiver>
</application>
```

In Java:

```
receiver = new BroadcastReceiver() { ... }

protected void onResume() {
    registerReceiver(receiver, new IntentFilter(Intent.ACTION_TIME_TICK));
}

protected void onPause() {
    unregisterReceiver(receiver);
}
```



onReceive() viene invocato quando l'evento registrato (l'evento che viene indicato che il broadcast receiver deve ascoltare) avviene. Il problema è che ??? di conseguenza per fare operazioni anche leggermente lunghe conviene farlo in un nuovo thread???

Data management:

esistono molti modi per salvare permanentemente dati in un dispositivo mobile, in questo corso si vedono in particolare:

- **shared preferences**: coppie chiave-valore non fatte per essere un database con grandi quantità di dati, ma servono per mantenere alcuni parametri di configurazione dell'app.
- **Files**: si possono salvare dati su file del file system (direttamente sul dispositivo o sulla scheda SD), il quale però dà un accesso limitato all'applicazione che non può vedere lo spazio delle altre.
- **Database SQLite**: questo DBMS è già presente nel sistema operativo e si può utilizzare per salvare database.
- **Content provider**: l'applicazione può condividere tramite un'interfaccia i propri dati con le altre applicazioni.

Shared preferences:

Sono coppie chiave-valore che fino ad android 7 potevano essere sia private che pubbliche, ora sono solo private quindi le può leggere solo l'applicazione.

Attualmente la loro visibilità può essere estesa a tutta l'applicazione o ridotta a delle specifiche activity.

Per ottenere le preferences si fa (da Android 7.0 in poi):

```
getSharedPreferences(String preference, Context.MODE_PRIVATE);
```

Se si fa la get di una preference che non esiste essa viene creata.

Esistono anche le **preferences di default** che sono presenti in ogni applicazione, per ottenerle si fa:

```
getDefaultSharedPreferences();
```

Nel caso le si vogliano modificare di deve utilizzare l'oggetto SharedPreferences.Editor, fare le modifiche e poi fare **commit()** (oppure fare apply() per fare il commit in maniera asincrona e non bloccare il thread):

```
pref = getActivity().getPreferences(Context.MODE_PRIVATE);
SharedPreferences.Editor editor = pref.edit();
editor.putString("myDataLabel", "myDataValue");
editor.commit();
```

Preferences screen:

I preferences screen sono dei fragment già precostituiti forniti dalla Settings API di Jetpack per far interagire l'utente con le preference.

Per costruire una schermata la si compone nel file res/xml:

```
<PreferenceScreen
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <SwitchPreferenceCompat
        app:key="notifications"
        app:title="Enable message notifications"/>
    <Preference
        app:key="feedback"
        app:title="Send feedback"
        app:summary="Report technical issues or suggest new features"/>
</PreferenceScreen>
```

Esistono tanti tipi diversi di preference con cui si possono comporre le schermate (per esempio: CheckBoxPreference, EditTextPreference, ListPreference, RingtonePreference, per l'elenco completo vedere la documentazione

<https://source.android.com/devices/tech/settings/settings-guidelines>).

Dal codice poi si richiama la schermata nella seguente maniera, in questo modo vengono modificate le default shared preferences:

```
public class MySettingsFragment extends PreferenceFragmentCompat {
    @Override
    public void onCreatePreferences(Bundle savedInstanceState, String rootKey) {
        setPreferencesFromResource(R.xml.preferences, rootKey);
    }
}
```

↑
Replaces preferences instead of adding to the current hierarchy

Manipolare i file in java:

Creare un file:

```
File file = new File(...);
file.createNewFile();
```

Scrivere su file:

```
FileOutputStream outputStream = openFileOutput("fileName.txt", Context.MODE_PRIVATE);
outputStream.write("Internal Content".getBytes());
outputStream.close();
```

Leggere da file:

```
FileInputStream inputStream = context.openFileInput("filename.txt");
InputStreamReader inputStreamReader =
    new InputStreamReader(inputStream, StandardCharsets.UTF_8);
StringBuilder stringBuilder = new StringBuilder();
BufferedReader reader = new BufferedReader(inputStreamReader) {
    String line = reader.readLine();
    while (line != null) {
        stringBuilder.append(line).append("\n");
        line = reader.readLine();
    }
}
String contents = stringBuilder.toString();
```

Files:

i file possono trovarsi nell'archiviazione interna del telefono oppure in una memoria esterna (tipo scheda sd) alla quale si accede attraverso librerie specifiche.

Le applicazioni invece possono salvare nella memoria interna del telefono nella loro sottocartella res dei file XML o RAW.

I file raw si trovano in res/raw e sono tutto ciò che non è immagine, file di testo o altro già supportato in altra maniera.

Vengono letti come stream di byte e sta all'app interpretarli.

I file xml invece sono file xml generici per contenere informazioni in maniera strutturata.

I file interni sono invece file sempre creati dall'app al suo interno e non sono nemmeno visibili dalle altre applicazioni e vengono salvati in una parte del file system dedicata all'app.

i file esterni invece non sono comunque visibili dalle altre app, ma vengono salvati in un'area comune visibile a tutti in /storage/emulated.

Adesso i file esterni vengono gestiti tramite un insieme di api a parte.

SQLite:

Difficilmente si interagisce in maniera "grezza" col database, ma di solito si usano una serie di framework come rooms??.

Esiste questa possibilità di avere un database in locale perchè non si può sempre assumere che ci sia connettività alla rete (altrimenti si dice che la rete è la SSOT, single source of truth).

è buona norma quindi avere un database locale che si sincronizza con uno online.

La maniera grezza e non più utilizzata per accedere al database è estendere

SQLiteOpenHelper con tutti i metodi necessari per gestire il database.

Oppure si può creare un contract: una classe finale che stabilisce determinati formati???.

Oppure ancora si può fare in modo di eseguire direttamente codice SQL:

❖ Constructor: call the superconstructor

```
Public mySQLiteHelper(Context context) {  
    super(context, DB_NAME, null, DB_VERSION);  
}  
DB_NAME could be  
"grades.db" in our example  
DB_VERSION = 1
```

❖ onCreate(SQLiteDatabase db): create the tables

```
String sql_grade = "create table " + TABLE_GRADES + "(" +  
    COL_ID + " integer primary key autoincrement, " +  
    COL_FIRSTNAME + " text not null, " +  
    COL_LASTNAME + " text not null, " +  
    COL_CLASS + " text not null, " +  
    COL_GRADE + " text not null );";  
db.execSQL(sql_grade);  
If using a contract, this is replaced by  
_ID
```

Per chiudere il database:

```
protected void onDestroy() {  
    sql.close();  
    super.onDestroy();  
}
```

Senza utilizzare il codice SQL ci sono modi migliore per interagire col database:

INSERT:

```
mySQLiteHelper sql = new mySQLiteHelper(getContext());  
SQLiteDatabase db = mySQLiteHelper.getWritableDatabase();  
  
ContentValues cv = new ContentValues();  
cv.put(mySQLiteHelper.COL_FIRSTNAME, firstName);  
cv.put(mySQLiteHelper.COL_LASTNAME, lastName);  
cv.put(mySQLiteHelper.COL_CLASS, className);  
cv.put(mySQLiteHelper.COL_GRADE, grade);  
long id = db.insert(mySQLiteHelper.TABLE_GRADES, null, cv);
```

DELETE:

- ❖ Best practice: Create a public method, like `deleteDb(...)`
- ❖ The delete method returns the number of rows affected
- ❖ Example:

```
db.delete(mySQLiteHelper.TABLE_GRADES, "id = ?", new String[]
{Integer.toString(id_to_delete)});
```

UPDATE:

```
ContentValues cv = new ContentValues();
values.put(mySQLiteHelper.FIRSTNAME, firstName);
values.put(mySQLiteHelper.LASTNAME, lastName);

db.update(mySQLiteHelper.TABLE_GRADES, values, "id = ?", new String[]
{Integer.toString(id_to_update)});
```

SEARCH (select):

```
Cursor gradeCursor = db.query(mySQLiteHelper.TABLE_GRADES, // the table (FROM)
new String[]{mySQLiteHelper.COL_GRADE}, // Cols to include in result (the SELECT)
mySQLiteHelper.COL_ID + " = ?",
new String[] {Integer.toString(id_to_update)}, // Inject in the where
null, // GROUP BY?
null, // FILTER BY?
mySQLiteHelper.COL_GRADE + " DESC"); // SORT BY?
```

ciò che viene ritornato da questa operazione è un cursor, ovvero un tabella con un puntatore che si può utilizzare per navigare in essa.

Il cursor ha una serie di metodi per navigare in esso e farne il parsing.

Content Provider:

I content provider sono un sistema per rendere disponibili dei dati alle altre applicazioni.

Il content provider dall'esterno viene visto come un database.

Ogni content provider ha uno o più uri.

L'esempio più classico di content provider è la rubrica del telefono, che mette a disposizione i contatti come se fossero in un database, oppure altri sono:

Class	Description
AlarmClock	To interact with the alarm
BlockedNumberContract	To get blocked numbers
Browser	To perform commands on the browser
CalendarContract	To handle calendar information
CallLog	Log of past calls
ContactsContract	Get and add contacts
DocumentsContract	Interact with documents
DocumentsProvider	Interact with documents
MediaStore	Access Video, Pictures, Audio and more
Settings	Inquiry system settings
...	

Attenzione: spesso per accedere ai content provider servono dei permessi.
I content provider che si espongono vanno inseriti nel manifest.

Per accedere ad un content provider si fa:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    Cursor mCursor = getContentResolver().query(
        ContactsContract.Contacts.CONTENT_URI, null, null, null, null);
    while (mCursor.moveToNext()) {
        String contactName = mCursor.getString(cursor.getColumnIndex(
            ContactsContract.Contacts.DISPLAY_NAME));
    }
    mCursor.close();
}
```

Invece per esporre un content provider:

```

public class ExampleProvider extends ContentProvider {
    private static final UriMatcher uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);

    static {
        uriMatcher.addURI("com.example.app.provider", "table3", 1);
        uriMatcher.addURI("com.example.app.provider", "table3/#", 2); } // Wildcard

    public Cursor query ( Uri uri, [[ usual query params ... ]] ) {
        switch (uriMatcher.match(uri)){
            case 1: ...
            case 2: ...
        }
    }
}

```

E lo si inserisce nel manifest utilizzando il tag <provider>, indicando android:authorities (il suo nome), android:name (la classe che lo implementa) ed i permessi di cui ha bisogno.

Per condividere interi file si utilizza la classe file provider:

```

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapp">
    <application
        ...
        <provider
            android:name="androidx.core.content.FileProvider"
            android:authorities="com.example.myapp.fileprovider"
            android:grantUriPermissions="true"
            android:exported="false">
            <meta-data
                android:name="android.support.FILE_PROVIDER_PATHS"
                android:resource="@xml/filepaths" />
        </provider>
        ...
    </application>
</manifest>

```

Inserendo i path dei file in android:resource, questi path devono essere creati in res/xml/filepaths.xml:

```

<paths>
    <files-path path="/" name="myimages" />
</paths>

```

Network:

Un'applicazione per andare su internet necessita di permessi specifici.

Le operazioni di rete sono costose per i dispositivi: in termini di tempo, batteria e costo del traffico dati.

Per il motivo del tempo è necessario fare le operazioni di rete su altri thread.

Per quanto riguarda la batteria ed i costi è necessario fare una serie di ottimizzazioni come mantenere una cache e posticipare le operazioni nei momenti migliori (il dispositivo è in carica, si è collegati al wifi, ecc...), oppure raggruppare le operazioni di rete e farle tutte insieme, in modo da far consumare meno l'interfaccia di rete.

Ottenere informazioni sulla connettività:

```
ConnectivityManager connMgr = (ConnectivityManager)
    getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
if (networkInfo != null && networkInfo.isConnected()) {
    // fetch data
} else {
    // display error
}
```

(prima di fare ogni operazione è necessario verificare se si è connessi ad internet).

Web view:

Le webview sono delle aree della nostra applicazione dove vengono caricate e renderizzate delle pagine web (possono essere anche pagine in locale).

Non presenta le caratteristiche tipiche di un browser come la barra degli indirizzi o la barra di navigazione, ad ogni modo le relative funzioni esistono e programmaticamente possono essere chiamate e sovrascritte:

- **public void goBack()**
- **public void goForward()**
- **public void reload()**
- **public void clearHistory()**

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {

    // Is there a page in the history?
    if ((keyCode == KeyEvent.KEYCODE_BACK) &&
        myWebView.canGoBack()) {
        myWebView.goBack();
        return true;
    }
    // Otherwise use the normal behavior
    return super.onKeyDown(keyCode, event);
}
```

Si può anche sovrascrivere il comportamento della web view quando si clicca sui link (per esempio per evitare di cliccare su cose che portano fuori dall'app):

```
private class MyWebViewClient extends WebViewClient {
    @Override
    public boolean shouldOverrideUrlLoading(WebView view, String url) {
        if ("www.mysite.com".equals(Uri.parse(url).getHost())) {
            // This is my website, so do not override; let my WebView load the page
            return false;
        }
        // The link is not for a page on my site, so throw the intent for browser
        Intent intent = new Intent(Intent.ACTION_VIEW, Uri.parse(url));
        startActivity(intent);
        return true;
    }
}
```

Inoltre ogni webview ha una serie di WebSettings copia...

Download Manager:

è un system service (service del sistema che è sempre running) che gestisce download di tipo http.

Si punta con un uri la risorsa da scaricare e quando il download manager ha finito viene mandato un broadcast intent.

Quindi le operazioni principali che si possono fare sono:

Data una richiesta:

```
Request request = new DownloadManager.Request(Uri.parse(address));
```

□ long enqueue(DownloadManager.Request)

```
long id = dm.enqueue(new DownloadManager.Request(uri)
.setAllowedNetworkTypes(DownloadManager.Request.NETWORK_WIFI |
    DownloadManager.Request.NETWORK_MOBILE)
.setDestinationInExternalPublicDir(Environment.DIRECTORY_DOWNLOADS,
    "output.txt"));
```

□ Cursor query(DownloadManager.Query)

```
Cursor c = dm.query(
    new DownloadManager.Query().setFilterById(id));
// can use DownloadManager.COLUMN_BYTES_DOWNLOADED_SO_FAR etc...
```

□ ParcelFileDescriptor openDownloadedFile(long) or better:

```
registerReceiver(myReceiver,
    new IntentFilter(DownloadManager.ACTION_DOWNLOAD_COMPLETE));
```

Classi HTTP:

Esistono due librerie più usate per fare connessioni HTTP tra app ed altri servizi:

- HttpClient (deprecata)
- HttpURLConnection (trattate in questo corso).

copia???

OKHttp:

libreria java di più alto livello rispetto a HttpURLConnection.

Si costruisce la richiesta:

```
OkHttpClient client = new OkHttpClient();
Request request = new Request.Builder()
    .url("https://www.unibo.it/sitoweb/federico.montori2")
    .build();
```

Oppure un po' più elaborata:

```
Request request = new Request.Builder()
    .header("Authorization", "your authorization here")
    .url("https://www.unibo.it/sitoweb/federico.montori2")
    .build();
```

E la chiamata si può fare sia in maniera sincrona (quasi mai usata):

```
Response response = client.newCall(request).execute();
```

che asincrona:

```
client.newCall(request).enqueue(new Callback() {  
    @Override  
    public void onFailure(Call call, IOException e) {}  
  
    @Override  
    public void onResponse(Call call, final Response response) {  
        if (response.isSuccessful()) {  
            // Here we have the response  
    }  
}
```

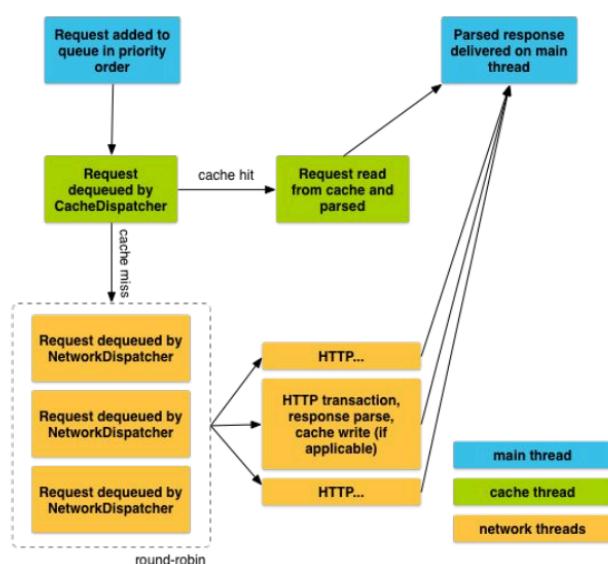
In questo caso sono richieste due funzioni di callback da chiamare quando si riceve una risposta (anche se contiene un messaggio di errore) o quando si ha un problema di altra natura.

WebSockets:

I websockets sono un modo di fare una comunicazione full duplex tra due dispositivi in rete. full-duplex significa che non c'è un rapporto client-server, ma entrambi i dispositivi possono comunicare nella stessa maniera alla pari.

Volley:

è una libreria per fare operazioni http e che ha molte features che gestisce autonomamente come caching e gestione dei thread.



Quando il client fa una richiesta viene prima verificato che essa sia stata cacheata e nel caso viene restituita la risposta che si ha in locale, altrimenti viene effettivamente fatta la richiesta.

Per aggiungere volley a gradle bisogna inserire in build.gradle:

```
implementation 'com.android.volley:volley:1.1.1'
```

Per fare una richiesta:

```
RequestQueue queue = Volley.newRequestQueue(this);
StringRequest stringRequest = new StringRequest(Request.Method.GET, baseUrl,
    new Response.Listener<String>() {
        @Override
        public void onResponse(String response) { // do something }
    }, new Response.ErrorListener() {
        @Override
        public void onErrorResponse(VolleyError error) { // do something }
});
queue.add(stringRequest);
```

Per avere degli header personalizzati:

```
{
    @Override
    public Map<String, String> getHeaders() {
        Map<String, String> params = new HashMap<String, String>();
        params.put("x-vacationtoken", "secret_token");
        params.put("content-type", "application/json");
        return params;
}
```

Comunicazioni TCP/IP:

copia...

Peer to Peer:

viene fatto per una questione di privacy ed anche per una questione di prestazioni (????).
un modo per farlo è usare il Wi-Fi Direct, ovvero uno dei due dispositivi assume il ruolo del router e l'altro il ruolo del client ed in questa maniera si comunicano dati senza appoggiarsi a terzi.

copia...

Altri tipi di comunicazione:

copia...

Kotlin:

è il **linguaggio di programmazione ufficiale** per **Android** dal 2019.

Nasce nel 2011 creato da jetbrains ed è open source dal 2012.

è un linguaggio ad inferenza di tipo (come python) anche se è staticamente tipato (una volta che una variabile ha un tipo non si può più cambiare).

Non si ha il ; a fine riga, ma è il carattere di andata a capo ad identificare la fine dell'istruzione.

Viene compilato in **bytecode**, quindi si esegue sulla stessa macchina virtuale di java ed è compatibile con ogni versione di android.

Differenza tra java e kotlin:



- Explicit types
- Strictly OOP
- Not Null Safe
- Explicit set & get



- Type inference
- Not necessarily OOP
- Null Safe
- Implicit set & get
 - + Extension functions
 - + Scope Functions
 - + Lambdas
 - + Implicit Casting
 - + Structured Concurrency
 - Coroutines (TBC)

Dichiarazione di variabili e tipi:

```
var x: Int = 42           // Declaration of a variable with type Int
var x = 42                // Declaration of a variable with inferred type Int
val x = 42                // Declaration of a constant with inferred type Int
```

Attenzione però, il tipaggio è statico:

```
var x = 42
x = 'c'                  // This will give an error
```

Come tipi primitivi si hanno:

- Int
- Long
- Short
- Byte
- Float
- Double
- Boolean
- Char
- String

Operatori:

Gli operatori in kotlin sono simili a quelli dei più comuni linguaggi di programmazione:

- Arithmetic Operators
 - + - * / %
- Logical Operators
 - && || !
- Comparison Operators
 - < > == >= <= !=

Punto di ingresso:

Kotlin ha come funzione principale il **main**:

```
fun main() {  
    val nickname: String = "stradivarius"  
    println("Hello world, my name is $nickname")  
}
```

Costrutti di selezione:

Il costrutto **if-then-else** è abbastanza standard:

```
if ( condition ) {  
    // Then Clause  
} else {  
    // Else Clause  
}
```

e può essere utilizzato anche negli assegnamenti:

```
var y = if (x == 42) 1 else 0
```

Invece il costrutto **case** si formula nella seguente maniera:

dove .. indica un **intervallo**.

Array:

```
when ( x ) {  
    in 0..21 -> println("One line clause")  
    in 22..42 -> println {  
        println("Multiple line clause")  
    }  
    else -> println("Default clause")  
}
```

```
val arr: Array<Int> = intArrayOf(1, 2, 3)           // [1,2,3]
println(arr[0])
```

Arrays are a class and can be instantiated in several ways (they also have their subtypes):

```
// Array of int of size 5 with values [0, 0, 0, 0, 0]
val arr = IntArray(5)
```

```
// Array of int of size 5 with values [42, 42, 42, 42, 42]
val arr = IntArray(5) { 42 }
```

```
// Array of int of size 5 with values [0, 1, 2, 3, 4] (lambda, you'll see...)
var arr = IntArray(5) { it * 1 }
```

Liste:

In Kotlin le liste sono generalmente strutture dati non mutabili e si creano utilizzando `listOf<Type>`.

Per avere delle liste modificabili si utilizza invece `mutableListOf<Type>`.

```
// Immutable List
val myList = listOf<String>("one", "two", "three")
println(myList[0])
```

```
// Mutable List (referenced by a val because it is the pointer)
val myMutableList = mutableListOf<String>("one", "two", "three")
myMutableList.add("four")
```

Costrutti di iterazione:

Anche i costrutti di iterazione (determinata e non) sono abbastanza standard:

```
// While loop
var counter = 0
while (counter < myMutableList.size) {
    println(myMutableList[counter])
    counter++
}

// For loop
for(item in myListMutable)      // Here we can use ranges as well
    println(item)
```

Null Safety:

Per definizione la sintassi del linguaggio è costruita in modo da evitare le NullPointerException a runtime.

Per questo motivo le variabili che possono assumere il valore null sono dette **nullable** (nullabili) e devono essere dichiarate come tali.

In questo modo quando si lavora con queste variabili si deve sempre **gestire esplicitamente** il caso in cui assumano il valore null (altrimenti si ha errore di compilazione).

Non nullable types

```
var s: String = "Hello" // Regular initialization means non-null by default  
s = null // compilation error
```

Nullable types

```
var s: String? = "Hello" // Nullable initialization means it can be null  
s = null // this is ok: e.g. if you print it, it will print "null"
```

Null safety

```
val l = s.length // Compiler error: "s can be null"  
val l = s?.length // If s is null then l is null (if nullable)  
val l = if (s != null) s.length else -1 // Custom workaround
```

Questo tipo di approccio ha alcune caratteristiche sintattiche particolari:

```
val name: String? = department?.head?.getName()  
name? = department.head.getName()
```

If anything in here is null, then the function is not called

You really want it to be not null:

```
val l = s!!.length // Casts s to non nullable, can throw exception
```

The “Elvis” operator

```
val l = s?.length ?: -1 // -1 is the default value for l if s is null
```

Funzioni:

le funzioni in kotlin hanno una sintassi del tipo:

```
fun isEven(number: Int = 0): Boolean { // number is set to 0 if not passed  
    return number % 2 == 0  
}  
isEven(14)
```

Mentre se sono utilizzate per estendere una classe hanno forma:

```

fun Int.isEven(): Boolean { // Extend the class Int
    return this % 2 == 0
}
14.isEven()

```

Funzioni di ordine superiore:

Higher order functions take functions as inputs

```

fun List<String>.customCount(function: (String) -> Boolean): Int {
    var counter = 0
    for (str in this) {
        if (function(str))
            counter++
    }
    return counter
} // Function that counts members in a List of strings that respect a certain condition

```

They might as well take any type in (usually called “generics”)

```

fun <T> List<T>.customCountAllTypes(function: (T) -> Boolean): Int {
    var counter = 0
    for (anything in this) {
        if (function(anything))
            counter++
    }
    return counter
} // Function that counts members in a List of any type that respect a certain condition

```

Lambda:

Sono funzioni senza nome perchè utilizzate solo una volta (esistono anche in java). Per esempio si possono passare delle funzioni lambda alle funzioni di ordine superiore definite nel paragrafo precedente:

```
val myList = listOf<String>("one", "two", "three")
```

```
val x: Int = myList.customCount { str -> str.length == 3 }
```

```
val x: Int = myList.customCountAllTypes { str -> str.length == 3 }
```

Classi:

simili a quelle di java ma hanno una sintassi diversa per il costruttore, il quale viene inserito direttamente nell'intestazione della classe.

```

class Animal (
    val name: String,
    val legCount: Int = 4
) {
    var sound: String = "Hey"           // Default value if not passed

    init {
        println("Hello I am a $name") // Function executed at instantiation time
    }
}
val dog = Animal("dog")            // Instantiation of a class into an object
val duck = Animal("duck", 2)

```

Inoltre tutti gli attributi hanno di default un **getter** ed un **setter** per potervi accedere dall'esterno.

Per sovrascrivere il getter o il setter si fa:

```

// Custom notation
var sound: String = "Hey"
    get() = this.name
    private set                         // Setter is private

```

Quindi facendo dog.sound non si accede direttamente al campo, ma viene chiamato il suo getter.

Esattamente come in java si possono fare sottoclassi e classi astratte.

```

class Dog: Animal("dog") {
    fun bark() {
        println("WOOF")
    }
}

class Duck: Animal("duck", 2) {
    fun quack() {
        println("QUACK")
    }
}

abstract class AbstractAnimal (
    val name: String,
    val legCount: Int = 4
) {
    abstract fun makeSound()
}

```

Classi anonime:

è possibile creare un oggetto di una classe che si scrive in quel momento solo per quell'istanza:

```

val bear = object: AbstractAnimal("bear") {
    override fun makeSound() {
        println("GROWL")
    }
}

```

Scope functions:

sono funzioni di linguaggio utilizzate per semplificare a livello di sintassi un insieme di operazioni sullo stesso oggetto.

Apply:

```
val snake = Animal("snake")           // Without "apply"  
snake.legCount = 0  
snake.sound = "Hiss"
```

```
val snake = Animal("snake").apply {    // With "apply"  
    legCount = 0  
    sound = "Hiss"  
}
```

In questo modo si omette lo "snake."

Esistono anche le funzioni di scope **let**, **with**, **run** e **also** con scopi simili.

Components Architecture:

In questo paragrafo si tratta dei design pattern da utilizzare nello sviluppo delle applicazioni android.

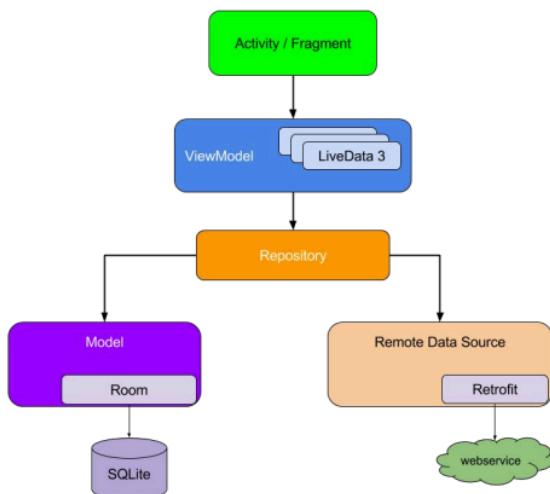
Android nel tempo è diventato molto difficile da gestire, tra diversi linguaggi nativi, tecnologie diverse, molto codice boilerplate e così via.

Inoltre tutti gli sviluppatori devono gestire tutto un insieme di problemi che si possono sempre presentare, come l'arrivo di una chiamata o il fatto che l'applicazione può andare in background.

Per semplificare questa situazione si cerca di tendere al massimo verso il **disaccoppiamento architetturale**: i vari componenti dell'app devono dipendere il meno possibile gli uni dagli altri in modo che i vari programmatore possano lavorare sui vari componenti nella maniera più autonoma possibile.

Model View ViewModel (MVVM):

è il principale design pattern promosso dalla google e da utilizzare nello sviluppo di app android.



Siccome eliminare completamente le dipendenze è impossibile (altrimenti le varie parti dell'app non comunicherebbero tra di loro), si propone per adottare questo **modello a cascata**: ogni elemento dello strato inferiore non dipende da ciò che ha sopra, mentre chi sta sopra dipende dallo strato inferiore.

Per fare ciò si utilizzano componenti e strutture dati specifiche, che vanno importati inserendo le seguenti dipendenze:

```
implementation "androidx.lifecycle:lifecycle-viewmodel:2.2.0"
implementation "androidx.lifecycle:lifecycle-livedata:2.2.0"
implementation "androidx.lifecycle:lifecycle-common-java8:2.2.0"
```

ViewModel:

è un componente che contiene tutti i dati relativi alla UI e parte della logica (la restante sta nelle activity), inoltre è LifeCycle Aware (vedi dopo).

Avendo il ViewModel tutti i dati relativi alla UI non è necessario salvare l'instance state ogni volta.

Per creare un ViewModel:

```
public class MyViewModel extends ViewModel {
    private List<User> users;
    public List<User> getUsers() {
        // Do an asynchronous operation to fetch users.
        return users;
    }
}
```

Per inizializzare invece un oggetto di tipo ViewModel non si può utilizzare l'operatore new, ma si deve fare:

```
MyViewModel model = new ViewModelProvider(this).get(MyViewModel.class);
List<User> users = model.getUsers();
```

Questo perché di ViewModel ne esiste uno solo per contesto, quindi sono una specie di singleton.

Attenzione: **un ViewModel non si riferisce mai direttamente a dati che si trovano nell'activity.**

Per leggere i dati dell'activity il ViewModel utilizza il design pattern observer (<https://refactoring.guru/design-patterns/observer>).

I LiveData sono dei dati osservabili che hanno una LifeCycle Awareness e che si devono tenere nel ViewModel.

LifeCycle Awareness indica che si sta implementando il LifecycleObserver, ovvero ha una serie di metodi che vengono chiamati quando ci sono dei cambiamenti nel suo ciclo di vita:

```

public class MyObserver implements LifecycleObserver {
    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
    public void function1() { ... }

    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
    public void function2() { ... }

}

myLifecycleOwner.getLifecycle().addObserver(new MyObserver());

```

Quindi i LiveData avendo già questo meccanismo hanno la capacità di notificare i loro osservatori solo se sono in stato Started o Resumed.

Per dichiarare un LiveData si fa:

```
private MutableLiveData<String> currentName;
```

Un MutableLiveData può essere castato a LiveData se non si vuole che venga più modificato.

Differenze tra MVVM e MVC:

Key differences are in different separation of concerns.



- Controller is the Active Part
- Easy to test Model
- Uneasy to test the Controller because it is tied heavily to the API and the View.
- If we change the View, we change the controller
- View is the Active Part
- Business Logic separated from UI
- ViewModel prepares observable data
- Easier to test components separately.
- Need DataBinding to fully unleash...

Room:

Room è una libreria che costruisce un livello di astrazione sopra a SQLite e che viene utilizzato per implementare il Model del MVVM.

Room serializza i dati in oggetti e crea un accoppiamento stretto tra le classi ed i dati nel database.

In room ci sono tre componenti fondamentali:

- database: classe che contiene il database
- entity: strutture dati

- Data Access Object (DAO): classi astratte in cui si indicano quali operazioni si possono fare sul database, queste operazioni non sono implementate a mano, ma room riesce a generare automaticamente il codice (??).

Database:

classe astratta che estende RoomDatabase e con un decorator indica cose aiuto mi sto perdendo.

Entity:

Retrofit:

libreria per fare chiamate http per ottenere dati strutturati, è type safe perchè serializza automaticamente i dati ricevuti (quindi non è necessario fare il parsing manualmente).

```
public class RetroPhoto {
    @SerializedName("albumId")
    private Integer albumId;
    @SerializedName("id")
    private Integer id;
    public RetroPhoto(Integer albumId, Integer id) {
        this.albumId = albumId;
        this.id = id;
    }
    //Setters and getters here...
}
```

Firebase:

è un database già implementato da google che ha sia una parte in locale che una in cloud.

è un database reattivo: si possono fare delle query osservabili.

è un database noSQL.

UI best practices (UI navigation):

Menu:

elemento che appare quando l'utente preme sul tasto menu (i tre puntini) nella toolbar (se presente).

è utile per dare diverse opzioni all'utente senza fargli lasciare l'activity.

i menù possono essere creati mettendoli in res/menu/ e facendo linflate nell'activity:

- ❖ [Create res/menu/menu.xml](#)
- ❖ [We need:](#)
 - ❖ IDs of menu elements
 - ❖ Title of each element
 - ❖ Icon of each element
- ❖ [Inside the Activity, create onCreateOptionsMenu\(\)](#)
 - ❖ Inflate the menu
 - ❖ Add functionality to the buttons

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
    <item android:id="@+id/item1" android:title="First Option"></item>
    <item android:id="@+id/item2" android:title="Second Option">
        <menu>
            <item android:id="@+id/item3" android:title="Third Option"/>
            <item android:id="@+id/item4" android:title="Fourth Option"/>
        </menu>
    </item>
</menu>
```

```
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);

    getMenuInflater().inflate(R.menu.myMenu, menu);

    // If you want to fire an intent when "menu_first" is pressed
    menu.findItem(R.id.menu_first).setIntent(new Intent(this, First.class));

    return true;
}
```

```
public boolean onOptionsItemSelected(MenuItem item) {
    switch ( item.getItemId() ) {
        case R.id.item1:
            /* do stuff */
            return true;
        [ ... ]
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

Oppure crearli direttamente tramite java dentro l'activity.

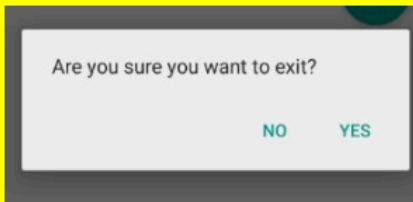
Snackbar:

sono un elemento grafico di notifica simile al toast ma con delle proprietà in più: la loro presenza è vincolata ad una view ed alcuni tipi di view ci interagiscono automaticamente, vi può essere fatto sopra swipe, può ascoltare determinati eventi e può avere dei pulsanti.

Dialog:

sono delle aree dello schermo che danno l'impressione di essere sopraelevate e che sono utilizzate per mostrare all'utente delle informazioni urgenti e che richiedono l'interazione dell'utente per poter andare avanti.

```
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setMessage("Are you sure you want to exit?").setCancelable(false);
builder.setPositiveButton("Yes", new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int id) {
        MenuExampleActivity.this.finish();
    }
});
builder.setNegativeButton("No", new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int id) {
        dialog.cancel();
    }
});
AlertDialog alert = builder.create();    alert.show();
```



Cancelable through back?

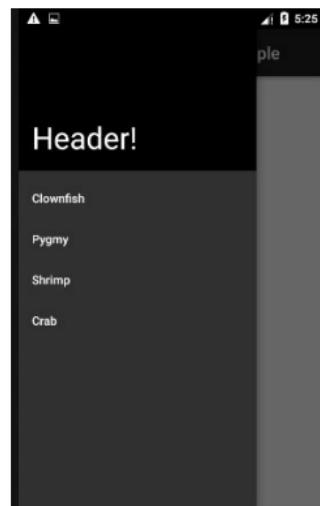
Per fare un dialog che ha un layout custom si deve chiamare `setView()` nel builder, con questo metodo si hanno ancora dei vincoli.

altrimenti per essere completamente liberi si deve estendere `FragmentDialog` e creare un nuovo fragment.

Navigation Drawer:

menù laterale che sta nascosto quando non si usa e può comparire con la pressione di un pulsante o facendo swipe da sinistra a destra.

I drawer layout diventa l'elemento padre del layout su cui si sta lavorando permettendo di avere il navigation drawer sempre presente e pronto a comparire.



Per creare un navigation drawer:

```
<androidx.drawerlayout.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android">
    <YourMainLayout ...> ... </YourMainLayout>

    <com.google.android.material.navigation.NavigationView
        ...
        app:headerLayout="@layout/MyHeader"
        app:menu="@menu/myMenu"
        ... />
</androidx.drawerlayout.widget.DrawerLayout>
```



Only valid for AndroidX
Otherwise the syntax is a bit different

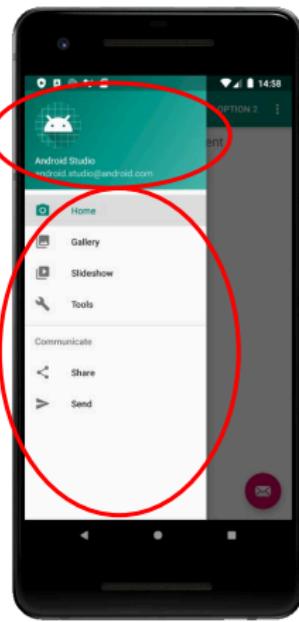
res/layout/myHeader.xml

```
<LinearLayout ...>
    <ImageView ... /> <TextView ... /> <TextView ... />
</LinearLayout>
```

res/menu/myMenu.xml

```
<menu
    xmlns:android="http://schemas.android.com/apk/res/android">
    <group android:checkableBehavior="single">
        <item ... /> <item ... /> <item ... /> <item ... />
    </group>
    <item android:title="Communicate">
        <menu>
            <item ... /> <item ... />
        </menu> </item>
    </menu>
```

Federico Montori - Programming with Android – Navigation



Anche in questo caso si possono ascoltare degli eventi:

```
NavigationView navigationView = findViewById(R.id.nav_view);

navigationView.setNavigationItemSelectedListener(
    new NavigationView.OnNavigationItemSelectedListener() {
        @Override
        public boolean onNavigationItemSelected(MenuItem menuItem) {
            menuItem.setChecked(true);
            mDrawerLayout.closeDrawers();

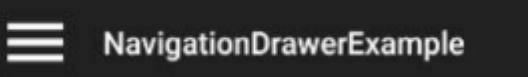
            ...
            return true;
        }
});
```

This is a very simple way...
- Use Navigation Framework
- NavController
- NavigationUI

Toolbar:

è un elemento grafico che sta anche esso fuori dal layout dell'activity e si trova sull'appbar (cosa in alto dove c'è il nome dell'app, dove si trova anche il menu).

copre la appbar e crea una cosa del genere:



Per creare una toolbar:

- Add the following inside the layout

```
<com.google.android.material.appbar.AppBarLayout android:theme="@style/AppTheme.AppBarOverlay">
    <androidx.appcompat.widget.Toolbar app:popupTheme="@style/AppTheme.PopupOverlay" />
</com.google.android.material.appbar.AppBarLayout>
```

- Set an appropriate theme in AndroidManifest.xml

```
    android:theme="@style/AppTheme"
```

- And in the Java class

```
Toolbar toolbar = findViewById(R.id.myToolbar);
setSupportActionBar(toolbar);
ActionBar actionBar = getSupportActionBar();
actionBar.setDisplayHomeAsUpEnabled(true);
actionBar.setHomeAsUpIndicator(R.drawable.ic_menu);
```

Stuff About the home icon...
in the next slide how to call it back.

Framework Navigation:

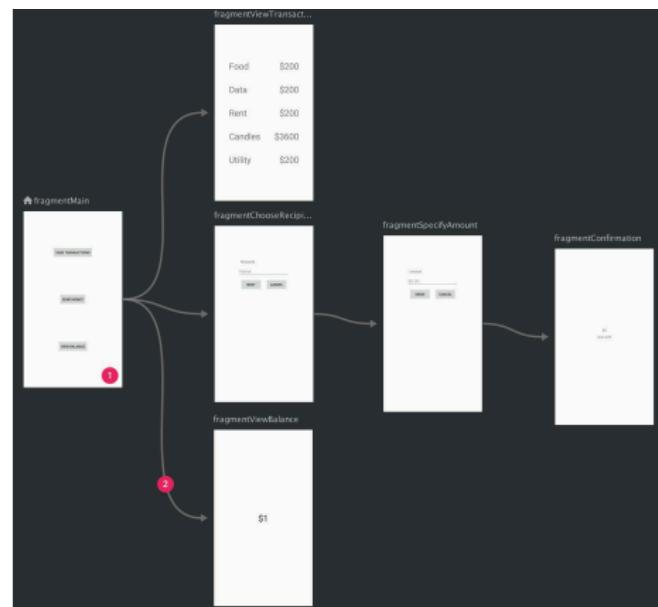
è un framework di jetpack per interagire con la navigazione in maniera più facile e dando al tutto un aspetto più uniforme.

la navigazione viene gestita tramite:

- **NavHostFragment** (in practice you have 1 Activity with many fragments interleaving in the NHF as container).
- **NavController** (the central brain)
- **A Navigation Graph**

The Navigation Graph:

- An XML resource connecting **destinations** (fragments) through **actions** (events).
- The XML resource type is “navigation”
- It must take place within a NavHostFragment (although destinations can also be activities).



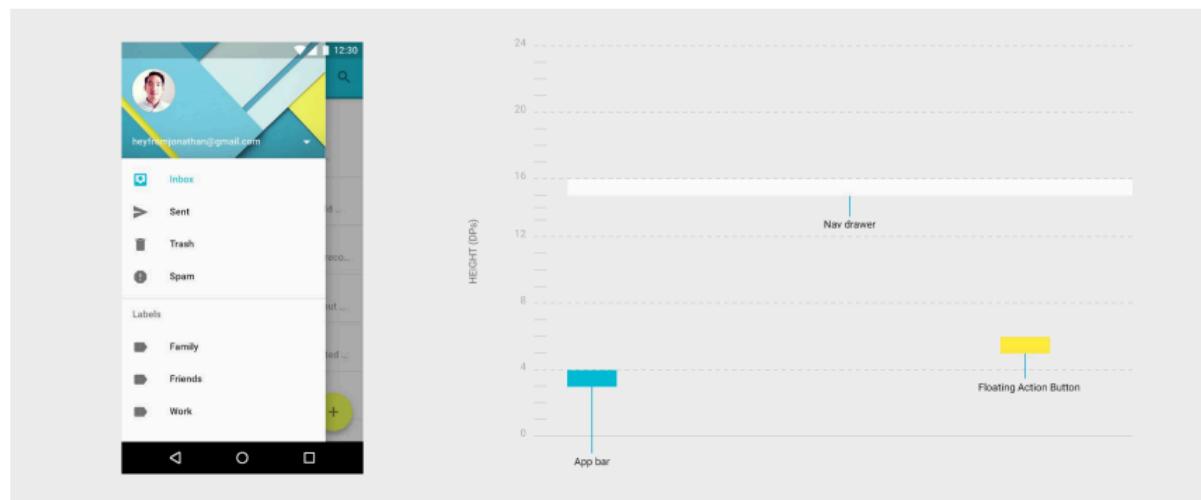
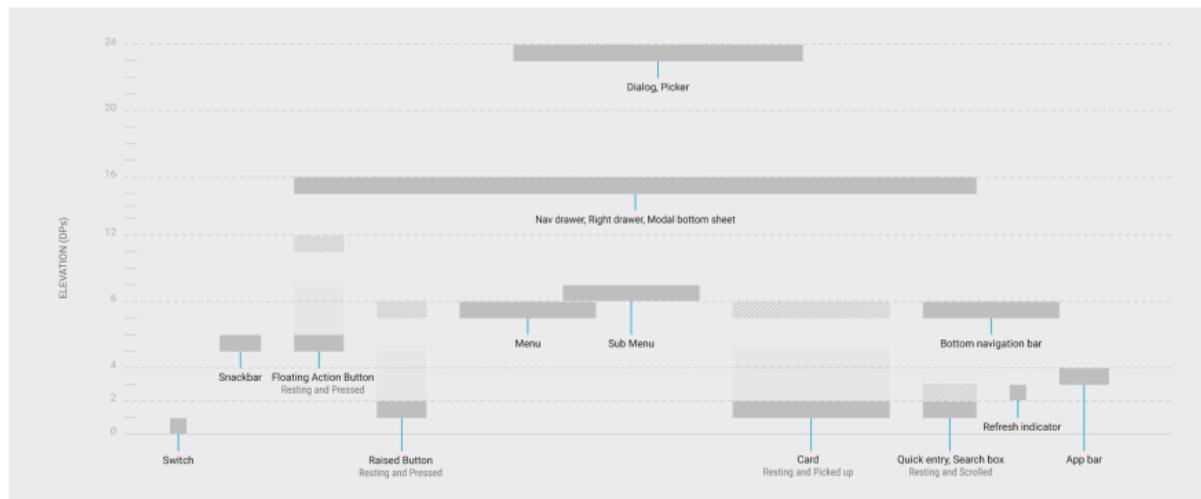
Questo grafo può essere modificato da android studio in via grafica tramite una schermata apposita (Navigation Editor) e viene tradotto poi in xml.
boh copia...

Material Design:

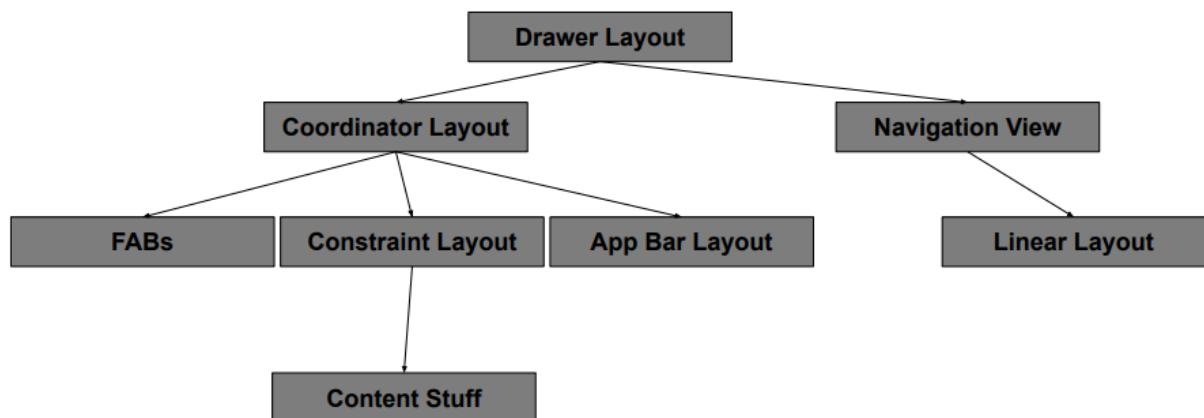
Insieme di linee guida ed elementi grafici proposti dalla google.

il material design copia i principi...

Ogni oggetto in material desing ha tre dimensioni: oltre alla posizione x ed y ha anche un'elevazione ed uno spessore.



In generale l'organizzazione della profondità degli elementi in una schermata rispetta questo schema, che è spesso lo stesso dell'organizzazione gerarchica dei layout:



Geolocalizzazione:

qui si utilizza google maps, ma esistono molti altri servizi di questo tipo.
la geolocalizzazione è uno dei fattori principali che rende le applicazioni mobili diverse da quelle desktop.

la geolocalizzazione è fatta tramite dei trasmettitori (gps e radio) e degli algoritmi di geolocalizzazione (l'hardware spesso non è sufficientemente preciso).
la geolocalizzazione è una delle caratteristiche che le applicazioni usano maggiormente per avere context awareness (capire in che contesto si è e cosa fare di conseguenza, si utilizza geolocalizzazione, orario, cosa sta facendo l'utente, con chi è, ecc...).

Il GPS (Global Positioning System) è un sistema che si basa su una flotta di satelliti chiamata Navstar (ma esistono anche altre flotte che si possono utilizzare) del dipartimento della difesa degli stati uniti.

Questi satelliti mandano periodicamente un segnale con la loro posizione ed il loro orario, il telefono riceve il segnale, calcola il ritardo con cui è arrivato il segnale ed è in grado di sapere circa la propria distanza dal satellite.

Ricevendo il segnale di almeno tre satelliti un dispositivo è in grado di triangolare la propria posizione.

In realtà si raggiunge un'accuratezza minimale con almeno quattro satelliti, questo perchè gli orologi dei satelliti e quello del dispositivo non è detto che siano perfettamente sincronizzati.

Si utilizzano tecniche simili per rilevare la propria posizione tramite il wifi, ma questa cosa è molto meno precisa.

Google per sapere dove sono gli access point wifi utilizza il gps di tutti gli utenti che si connettono e cerca di stimarne la posizione.

In android esiste il location manager che gestisce tutto questo aspetto e che manda semplicemente delle notifiche quando la posizione cambia.

Per usare la localizzazione esistono tre tipi di permessi:

- ◆ **ACCESS_FINE_LOCATION**
 - Allows the app to use any possible way to retrieve the location.
- ◆ **ACCESS_COARSE_LOCATION**
 - Allows the app to use only location data coming from wifi / cellular localization.
- ◆ **ACCESS_BACKGROUND_LOCATION**
 - To be requested in **addition** if you target API 29 or higher.
Here is an elaborated article on how:
<https://developer.android.com/training/location/request-updates#request-background-location>

Esistono due modi per gestire la geolocalizzazione nelle proprie app:

- andriod.location: è il metodo più vecchio e più semplice, ma consuma moltissima energia.
- Location services.

android.location:

si richiedono al location manager degli update periodici, il location manager si segna l'intervallo temporale richiesto dall'app e periodicamente glielo comunica.
il problema è quando più applicazioni richiedono la geolocalizzazione perchè le varie letture non possono essere fuse tra di loro e quindi vengono fatti tantissime letture per soddisfare tutti.
(copia come si fa a livello di codice).

FusedLocationProvider:

In questo caso le richieste delle varie applicazioni vengono fuse il più possibile, diminuendo la precisione temporale ma migliorando molto le prestazioni.

❖ Obtain the FusedLocationProviderClient (onCreate)

```
FusedLocationProviderClient mFusedLocationClient =
    LocationServices.getFusedLocationProviderClient(this);
```

❖ Get the location

```
mFusedLocationClient.getLastLocation()
    .addOnSuccessListener(this, new OnSuccessListener<Location>() {
        @Override
        public void onSuccess(Location location) {
            if (location != null) {
                // do something
            }
        }
    });
});
```

Per gli update:

❖ Create a LocationRequest

```
LocationRequest mLocationRequest = LocationRequest.create();
mLocationRequest.setInterval(10000);
mLocationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
```

❖ Location is then updated every 10 seconds

❖ Get it with getLastLocation()

Google Maps:

è il sistema maggiormente compatibile con Android (per ovvie ragioni), ma non è l'unico.

Alcune alternative sono:

- Mapbox: <https://docs.mapbox.com/android/maps/guides/>
- OsmDroid: <https://github.com/osmdroid/osmdroid>

Esistono due tipi di integrazione delle mappe:

- versione 1: una webview che mostra google maps, non è più utilizzata
- versione 2: fragment dedicato alle mappe.

per utilizzare google maps bisogna:

installare google play service sdk
ottenere una chiave valida da <https://cloud.google.com/console/google/maps-apis/overview>
inserire la chiave nell'app:

```
<meta-data
    android:name="com.google.android.geo.API_KEY"
    android:value="API_activation_key">

<permission
    android:name="com.example.mapdemo.permission.MAPS_RECEIVE"
    android:protectionLevel="signature"/>
<uses-permission
    android:name="com.example.mapdemo.permission.MAPS_RECEIVE"/>
<uses-feature
    android:glEsVersion="0x00020000"
    android:required="true"/>
```

aggiungere un MapFragment:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment
    android:id="@+id/map"
    android:name="com.google.android.gms.maps.SupportMapFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

e poi a livello di programmazione si gestisce il tutto utilizzando l'oggetto GoogleMap:

```
private GoogleMap mMap;
...
mMap = ((SupportMapFragment) getSupportFragmentManager()
    .findFragmentById(R.id.map)).getMap();
```

copia...

Google Direction API:

Non è un servizio dedicato, ma è un servizio web al quale si fanno delle richieste http e si ottiene una risposta sotto forma di json.

nella risposta c'è un percorso codificato con un algoritmo proprietario per cui per decodificarlo ed ottenere le coordinate è necessario utilizzare una libreria specifica.

GeoCoding:

la classe Geocoder permette di ottenere delle liste di indirizzi ai quali sono vicino (la geolocalizzazione non è precisa) o degli indirizzi che ho cercato con un nome (tipo la ricerca su google maps).

Geofencing:

metodo con il quale l'app riceve delle notifiche solo quando l'utente interagisce con una certa area di interesse circolare.

si possono avere fino a 100 geofence (zone di interesse).

attenzione: le geofence sono abbastanza dispendiose per quanto riguarda la batteria.

System Services:

Componenti governati dal sistema operativo a cui l'utente può richiedere dei servizi.

Alcuni di esse sono:

AccessibilityManager	JobScheduler	SensorManager
AccountManager	KeyguardManager	StorageManager,
ActivityManager	LauncherApps	SubscriptionManager
AlarmManager	LayoutInflator	TelecomManager
AppOpsManager	LocationManager	TelephonyManager
AudioManager	MediaProjectionManager	TextServicesManager
BatteryManager	MediaRouter	TvInputManager
BluetoothManager	MediaSessionManager	UiModeManager
ClipboardManager	MidiManager	UsageStatsManager
ConnectivityManager	NetworkStatsManager	UsbManager
DevicePolicyManager	NfcManager	UserManager
DisplayManager	NotificationManager	Vibrator
DownloadManager	NsdManager	WallpaperService
DropBoxManager	PowerManager	WifiManager
FingerprintManager	PrintManager	WifiP2pManager
InputMethodManager	RestrictionsManager	WindowManager
InputManager	SearchManager	

Sono componenti analoghi ai service che alle richieste danno spesso risposte in maniera asincrona, quindi spesso è necessario creare dei listener a loro dedicati.

Battery manager:

è il service che fornisce tutte le informazioni relative alla batteria.

Alarm Service:

prende un intent in input e lo lancia ad un determinato orario che gli viene passato sempre in input.

Attenzione: quando il telefono viene spento e riacceso tutti gli alarm vengono persi.

WorkManager:

è un'API che utilizzando l'alarm manager permette di schedulare delle attività asincrone (ma non ha alta precisione, quindi se serve lanciare qualcosa in un momento preciso bisogna usare a mano l'alarm manager).

Sensor Service:

è il servizio che permette di accedere a tutti i sensori del telefono.

Non sono necessari permessi speciali per accedere a tali sensori.

i sensori si classificano in tre macro categorie:

- sensori di movimento
- sensori di ambiente
- sensori di posizione

Accelerometro: sensore che fornisce l'accelerazione del telefono sui tre assi.

Giroscopio: misura l'orientamento del dispositivo sui tre assi e fornisce la velocità angolare del dispositivo.

Sensore di luce: utilizza dei fotodiodi.

Sensore di prossimità: usa infrarosso o ultrasuoni, è utile per capire se c'è qualcosa vicino allo schermo (tipo se si fa una telefonata spegnere lo schermo quando il volto è vicino).

E tanti altri:

❖ **public List<Sensor> getSensorList(int type);** (can be Sensor.TYPE_ALL)

Sensor	Type (Hardware/Software)	Used for
TYPE_ACCELEROMETER	Hardware	Acceleration along three axes (+ gravity)
TYPE_AMBIENT_TEMPERATURE	Hardware	Temperature
TYPE_GRAVITY	Can be both	Motion Detection
TYPE_GYROSCOPE	Hardware	Rotation
TYPE_LIGHT	Hardware	Ambient brightness
TYPE_LINEAR_ACCELERATION	Can be both	Acceleration along three axes (no gravity)
TYPE_MAGNETIC_FIELD	Hardware	Compass, indoor navigation
TYPE_ORIENTATION	Software	Obtaining device position
TYPE_PRESSURE	Hardware	Obtaining the height from sea level
TYPE_PROXIMITY	Hardware	Setting off the screen
TYPE_RELATIVE_HUMIDITY	Hardware	Humidity
TYPE_ROTATION_VECTOR	Can be both	Motion and Rotation detection

Un sensore software è una funzione che aggrega dati di altri sensori, li elabora e restituisce un risultato.

I sensori vengono utilizzati spesso per fare activity recognition.

Google mette a disposizione per fare ciò una sua libreria (activity recognition API) che però è un po' complessa a livello di permessi e non è accuratissima.

IOS

il sistema operativo si divide in: (componi questi appunti e gli screenshot delle slide)

core OS:

- è il kernel ispirato a BSDX (cerca)
- ha una componente di power management che è molto importante nei dispositivi mobili
- varie componenti di sicurezza

Core Services: astrazione successiva con strutture dati complesse

- preferences: insieme di configurazioni che indicano informazioni sulle impostazioni dell'utente

Media: l'iphone è stato il primo dispositivo che ha avuto un'integrazione davvero efficace dei vari media nei dispositivi mobili

- gestione degli ambienti smart
- gestione della realtà virtuale ed aumentata
- gestione della grafica 2d e 3d (quindi anche per il gaming)

Cocoa Touch: insieme delle librerie e processi che gestiscono l'interazione con l'utente

- gerarchia delle viste: si possono sovrapporre più visualizzazioni su una stessa schermata con vari livelli di opacità e sfocatura
- localizzazione linguistica: quali sono le lingue utilizzate dall'utente? che lingua mostrare?
- web view: browser embedded in una porzione dell'applicazione
- controllo della fotocamera ed altre periferiche

Piattaforma di programmazione:

i programmi per IOS non si realizzano su IOS.

L'SDK si chiama Xcode.

Il linguaggio è Swift, mentre il vecchio objective-C non è più supportato.

I framework sono estremamente ricchi.

Design Strategy: inizialmente era richiesto lo sviluppo secondo il pattern Model View Controller (tuttora consigliato), ora è consentito anche lo sviluppo secondo Model View Model.

MVC (model view controller):

il mondo Apple forza i programmatori a sviluppare le applicazioni in una certa maniera, questa cosa è spesso percepita come fortemente limitante ma permette un'ottimizzazione elevata per l'hardware sottostante.

Il pattern che deve essere seguito per sviluppare i programmi è il Model View Controller, nel quale gli elementi del programma sono divisi in tre gruppi:

- **model**: area che contiene i dati che rappresentano lo stato dell'esecuzione dell'applicazione (variabili, strutture dati, oggetti, ecc...), ma non informazioni su come queste cose sono mostrate a schermo; questo significa che le elaborazioni del programma sono fortemente disaccoppiate dalla loro visualizzazione.

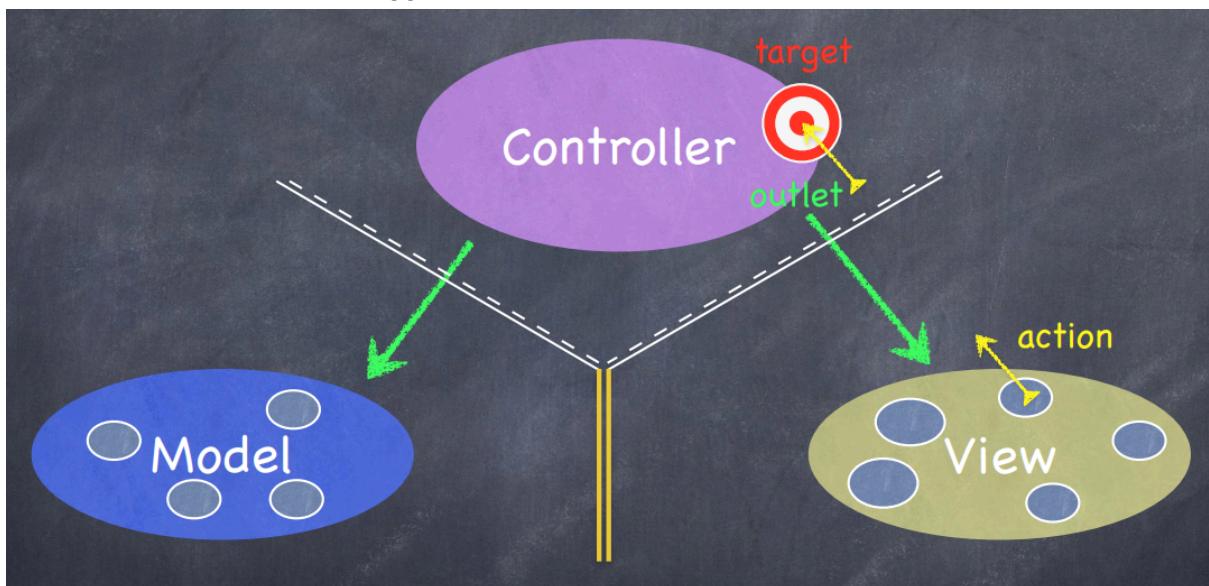
- **view**: elementi che mostrano all'utente qualcosa sia a livello grafico che in formato audio o altro, che cosa devono mostrare è invece gestito dal controller.
- **controller**: luogo dove si trova la logica del programma e le politiche di visualizzazione dei dati (UI logic).

Queste tre aree comunicano tra di loro in una maniera strutturata e gerarchica: il controller può mandare informazioni al model ed alla view ed eseguire il codice in essi contenuto, ma non viceversa; inoltre model e view non possono comunicare direttamente.

Se si hanno comunicazioni che non seguono queste linee guida si ha un warning.

La comunicazione tra controller e view avviene tramite un **outlet**, ovvero un puntatore alla view.

Esiste anche una comunicazione inversa dalla view al controller (per esempio per la ricezione degli input): il controller espone dei metodi **target** ai quali le view possono mandare delle **action** (messaggi con parametri).



Delegates (delegati):

sono metodi invocati dal sistema ogni volta che si verifica una determinata condizione.

Questi metodi sono implementati dal controller tramite i protocols (protocolli) che sono una definizione di una collezione di metodi senza la classe associata (tipo le interfacce). I delegati iniziamo con parole del tipo "will, should, did".

Data source:

sono metodi che permettono la comunicazione tra view e modello passando dal controller.

è possibile anche che il modello cambi senza l'azione del controller (tipo un buffer che si riempie in una comunicazione di rete), in questo caso c'è un sistema di message passing per notificare il controller (approfondisci).

Swift:

mi sono persa un sacco di cose fatte durante la demo, non so se le recupererò 😞.

Range:
ma che è?

Tuple:
una tupla è un raggruppamento di valori che si può utilizzare come tipo.
si fa utilizzando delle parentesi tonde:
`let x: (String, Int, Double) = ("hello", 5, 0.85)`

Computed properties:
una computed property è una variabile con un get o un set sovrascritti.
Queste variabili sono molto comode come campi delle classi perchè se ne può personalizzare il comportamento quando viene istanziato un oggetto.

Access Control:
ovvero la protezione dell'accesso ai campi delle classi.
Swift ha questi tipi di visibilità dei campi:

Extensions:
si possono estendere classi/struct/enum aggiungendovi metodi e campi.
non si possono però reimplementare metodi già presenti, ma se ne possono solo aggiungere di nuovi.
per quanto riguarda i campi si possono aggiungere ma non possono contenere un valore (non hanno un'area di memoria associata) e quindi di solito si utilizzano le computed properties per far calcolare il valore sul momento dal getter.

Enum:
un enumerazione (Enum in swift) è un tipo che può assumere uno degli stati specificati al suo interno.
Agli stati dell'Enum si possono associare anche degli attributi contenuti in essi (sia con nome che anonimi).
un parametro di tipo Enum è sempre passato per valore.
Le enum possono avere anche dei metodi e dei campi (sempre però computed properties).
Se vuole fare in modo che un metodo modifichi il valore della Enum si deve anteporre al metodo la parola chiave mutating (questo è perchè essendo un oggetto passato per valore bisogna aggiornare anche le altre copie???).

Optionals:
ogni oggetto di tipo optional può essere di un certo tipo oppure non definito.
Un optional è implementato come un'enum del tipo:

Gli optional inoltre hanno una sintassi particolare:

```
The "not set" case has a special keyword: nil  
The character ? is used to declare an Optional, e.g. var indexOfOneAndOnlyFaceUpCard: Int?  
The character ! is used to "unwrap" the associated data if an Optional is in the "set" state ...  
e.g. let index = cardButtons.index(of: button)!  
The keyword if can also be used to conditionally get the associated data ...  
e.g. if let index = cardButtons.index(of: button) { ... }  
An Optional declared with ! (instead of ?) will implicitly unwrap (add !) when accessed ...  
e.g. var flipCountIndex: UILabel! enables flipCountIndex.text = "" (i.e. no ! here)  
You can use ?? to create an expression which "defaults" to a value if an Optional is not set ...  
e.g. return emoji[card.identifier] ?? "?"  
You can also use ? when accessing an Optional to bail out of an expression midstream ...  
this is called Optional Chaining  
we'll take a closer look at it in a few slides
```

Gli optional vengono utilizzati per proteggere il programma da crash a runtime derivati da puntatori a null, infatti eliminano sintatticamente questa possibilità.

Classi:

Come nei principali linguaggi oo ci sono le classi.

ogni oggetto di una classe viene passato per puntatore.

Grazie al reference counting avviene automaticamente la deallocazione della memoria (ma non è garbage collection).

Esistono tre keyword da apporre alle variabili che influenzano il modo in cui viene fatto il reference counting:

- strong: modo di default, finchè esiste un puntatore a quell'area di memoria essa non viene eliminata.
- weak: se non ci sono puntatori non weak a quella variabile essa viene eliminata e viene ritornato nil (sostanzialmente sono puntatori che non vengono contati nel reference counting).
- unowned: il reference counting non viene fatto e sta al programmatore gestire il tutto (altamente sconsigliato, fatto per compatibilità con objective-c).

Struct:

tipi contenenti più campi e passati per valore.

Protocolli:

Collezione di diciture di metodi (è un tipo senza campi), è come un'interfaccia di java.
servono per raggruppare metodi che poi le classi possono implementare (estendendo il protocollo)

I protocolli vengono utilizzati per fare i delegates.

alcuni metodi nei protocolli possono essere indicati come opzionali (@objc) e significa che le classi possono anche non implementarli.

I protocolli possono estendersi a vicenda con una sorta di ereditarietà multipla:

```

protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}

```

Anyone that implements SomeProtocol must also implement InheritedProtocol1 and 2

copia mutating che non ho capito

anche struct ed enum possono implementare i protocolli.

Una classe per implementare dei protocolli li elenca subito dopo la superclasse:

```

class SomeClass : SuperclassOfSomeClass, SomeProtocol, AnotherProtocol {
    // implementation of SomeClass here
    // which must include all the properties and methods in SomeProtocol & AnotherProtocol
}

```

oppure si può fare esplicitamente nella seguente maniera:

```

extension Something : SomeProtocol {
    // implementation of SomeProtocol here
    // no stored properties though
}

```

Stringhe:

una stringa in swift è un array di caratteri unicode.

Siccome gli accenti alle volte sono salvati come caratteri a parte, l'indice della stringa non è un normale intero, ma è un oggetto di tipo Index che gestisce queste eccezioni uniformando le parole indipendentemente da come sono salvate.

```

let pizzaJoint = "café pesto"
let firstCharacterIndex = pizzaJoint.startIndex // of type String.Index
let fourthCharacterIndex = pizzaJoint.index(firstCharacterIndex, offsetBy: 3)
let fourthCharacter = pizzaJoint[fourthCharacterIndex] // é

if let firstSpace = pizzaJoint.index(of: " ") { // returns nil if " " not found
    let secondWordIndex = pizzaJoint.index(firstSpace, offsetBy: 1)
    let secondWord = pizzaJoint[secondWordIndex..

```

Una stringa è una struct immutabile, di conseguenza ogni volta che la si modifica in realtà viene creata una copia e riassegnata.

NSAttributedString:

sono stringhe con degli attributi aggiuntivi come il colore ed il font.

```

let attributes: [NSAttributedStringEncoding : Any] = [ // note: type cannot be inferred here
    .strokeColor : UIColor.orange,
    .strokeWidth : 5.0      // negative number here would mean fill (positive means outline)
]
let attribtext = NSAttributedString(string: "Flips: 0", attributes: attributes)
flipCountLabel.attributedText = attribtext // UIButton has attributedTitle

```

Attenzione: come eredità da objective C si ha che le NSAttributedStrings non sono modificabili a runtime, se se ne vuole avere una modificabile si deve creare una NSMutableAttributedString.

Tipi funzione:

Le funzioni sono implicitamente associate ad un tipo.

si può quindi associare ad una variabile una funzione:

```

Example ...
var operation: (Double) -> Double
This is a var called operation
It is of type "function that takes a Double and returns a Double"
You can assign it like any other variable ...
operation = sqrt // sqrt is just a function that takes a Double and returns a Double
You can "call" this function using syntax very similar to any function call ...
let result = operation(4.0) // result will be 2.0

```

da qui deriva la possibilità di creare delle closure, ovvero fare delle funzioni inline. (copia...)

Casting:

esiste il casting...

se i due tipi sono compatibili viene fatta la conversione, invece se due valori non hanno algoritmi di conversione il cast non funziona e viene ritornato nil.

Disegnare su schermo:

Views:

una view è un'area rettangolare disegnabile.

le view sono organizzate in gerarchia partendo dalla UIView, che è l'area più grande possibile e che comprende anche la barra di stato.

copia metodi per aggiungere subview e toglierle.

IBRIDI

In questo corso si studiano in particolare angular con le librerie ionic e capacitor.
le tecnologie ibride si basano su tecnologie web ed accedono ai componenti del dispositivo tramite librerie specifiche.

hanno prestazioni leggermente inferiori rispetto alle applicazioni native, ma permettono di passare dal mondo web ad i vari dispositivi in maniera molto facile.
un tipo particolare di app ibride sono le progressive webapp (PWA), ovvero web app adattate per eseguirsi su dispositivi mobili.

la UI è definita tramite HTML5, mentre lo scripting è fatto in javascript utilizzando tutti i più comuni framework.

le applicazioni native si dividono in:

- hybrid native: la ui viene ricostruita con strutture native del sistema operativo, in questo modo si ha più efficienza ma si è più limitati della personalizzazione della schermata.
- hybrid web: la UI è costruita in HTML e viene renderizzata a runtime, per accedere ad i componenti del dispositivo si utilizzano dei plugin.

Angular 2+:

molto diverso da angular.js (angular 1).

è un framework per lo sviluppo di applicazioni single page, ovvero che si basano su una singola pagina e la navigazione avviene tramite il proprio motore.

angular è ispirato al MVC ed il suo sistema si chiama Component based.

Non ha controller, ma ha dei componenti che sono fatti di template (le view).

Ionic:

Ionic è un framework open source per lo sviluppo della ui mobile.

Può interagire con tutti i principali framework (angular, vue, react, ecc...).

i componenti sono scritti in html con qualche tag personalizzato, lo stile viene automaticamente adattato allo stile del sistema operativo che si sta utilizzando.

Capacitor:

nasce come successore di apache cordova e phoneGap.

è un framework open source per lavorare con i sensori del dispositivo.

React Native:

React è un framework di meta creato per fare siti frontend.

React native invece fa la stessa cosa per le applicazioni.

Questo porta ad avere molti vantaggi come essere cross platform, ha molti plugin di terze parti e ci vogliono meno risorse per sviluppare.

è comodo per sviluppare applicazioni mobili perché si può utilizzare sia su android che su apple.

React ha una struttura a componenti ed utilizza javascript come linguaggio di programmazione.

La parte di javascript viene interpretata dal JavaScriptCore che fa eseguire dei pezzi di codice nativo.

copia tutto dalle slide.