Sviluppo Applicazioni Mobili

Federico Matteoni

A.A. 2019/20

Indice

Introduzione

Vincenzo Gervasi, `gervasi@di.unipi.it`
`circe.di.unipi.it/ gervasi/main/`
`developer.android.com`

Modalità d'esame Sviluppo di un'app, proposta dallo studente ma concordata con il docente. Presentazione dell'app con ispezione del codice e domande "teoriche" su aspetti non coperti dal progetto. No compitini.
3 criteri: applicazione mobile, non deve avere senso su applicazione web o su computer. diversità, almeno tre framework presentati. progetto adeguato a esame di 6 CFU.

Capitolo 1

Programmazione Android

1.1 Breve Storia di Android

2007 Telefonini Nokia, Palm, Windows CE e BlackBerry. Tutti **sistemi fortemente proprietari**, spesso con versioni frammentate e di difficile manutenzione. Giravano su una versione di Java portatile ma fortemente limitata, **JavaME**.

Novembre 2007 La **Open Handset Alliance**, formata da vari produttori di telefoni, pubblica la **Open Platform for Mobile Handset**.
Era il 5 Novembre, **7 giorni dopo rilasciano Android**.
Chi c'era dietro, tra le altre: Google, eBay, China Mobile, HTC, Intel, LG, Motorola, NTT DoCoMo, Qualcomm, nVidia, Samsung, Sprint Nextel, Telecom Italia, Telefònica, Texas Instrument, T-Mobile. Ovvero vari produttori di telefoni, di chip, fornitori di servizi e di telefonia.

Android Il 12 Novembre viene rilasciato **Android**
Rilasciato su licenza Apache, **basato su Linux 2.6** e sviluppato su Eclipse, Java e Python. Il kernel **era completo e standard**, non era personalizzato.
Sviluppato da **Android Inc.**, startup californiana nata nel 2003 a Palo Alto, acquistata da Google nel 2005 e brevetti registrati nel 2007. Lo sviluppo è avvenuto in gran segreto, brevetti registrati all'ultimo così da non destare sospetti a Microsoft e Apple. Fondata da **Andy Rubin**.

Adesso Dal 2007 sono state rilasciate numerose **versioni**, dai *codename* ispirati a nomi di dolciumi in ordine alfabetico... fino ad Android Q. Adesso parleremo principalmente di software, ma non bisogna dimenticare il lato **hardware**: potenza di calcolo, efficienza della batteria, sensori e schermi. I produttori, inoltre, hanno **poco interesse ad aggiornare i telefoni vecchi**: il principale problema è che per ogni modello e ogni aggiornamento bisogna far omologare e convalidare la parte telefonica, quindi servono mesi di test e tanti soldi. Per cui è meglio **spingere gli utenti a comprarne di nuovi** $\Rightarrow$ ***frammentazione***.

Software Ogni versione è (*quasi*) sempre **pienamente compatibile con le precedenti**: i cambiamenti nelle API sono identificati da un **API Level**.
Le applicazioni possono quindi dichiarare:

API Level minimo di cui hanno bisogno per funzionare

API Level targe per cui sono state scritte

API Level massimo oltre il quale non funzionano più (pessima idea, sconsigliato, obsoleto e ignorato già da Android 2.0.1)

I vincoli vengono verificati dal market e dalle procedure di aggiornamento del S.O.

Rispetto iOS, i quali dispositivi vengono (*quasi*) sempre aggiornati alla versione più recente, Android tende a diffondere gli aggiornamenti più lentamente: l'Android più recente è sempre una nicchia.

Supporto Google cerca di supportare più o meno all'infinito le vecchie versioni del S.O. con le **librerie di compatibilità** (`libcompat`).

Codice che le applicazioni possono includere nel loro "eseguibile"

Simula le funzioni delle versioni più recenti sulle versioni più vecchie

Inoltre, parte delle funzioni del S.O. sono incorporate nei **Google Play Services**, libreria aggiornabile dal market.
Un **grosso ostacolo** è la **customizzazione** (skinning) del sistema.

1.2 Ant e Gradle

Ant antiquato
Gradle più moderno.

Gradle Sistema di build avanzato, configurabile. Distribuito nel senso di risorse per lo sviluppo sparse in rete tramite URL. Fill-in del manifest (manifest contiene metadati per il s.o.), gradle genera e mantiene aggiornato il manifest.

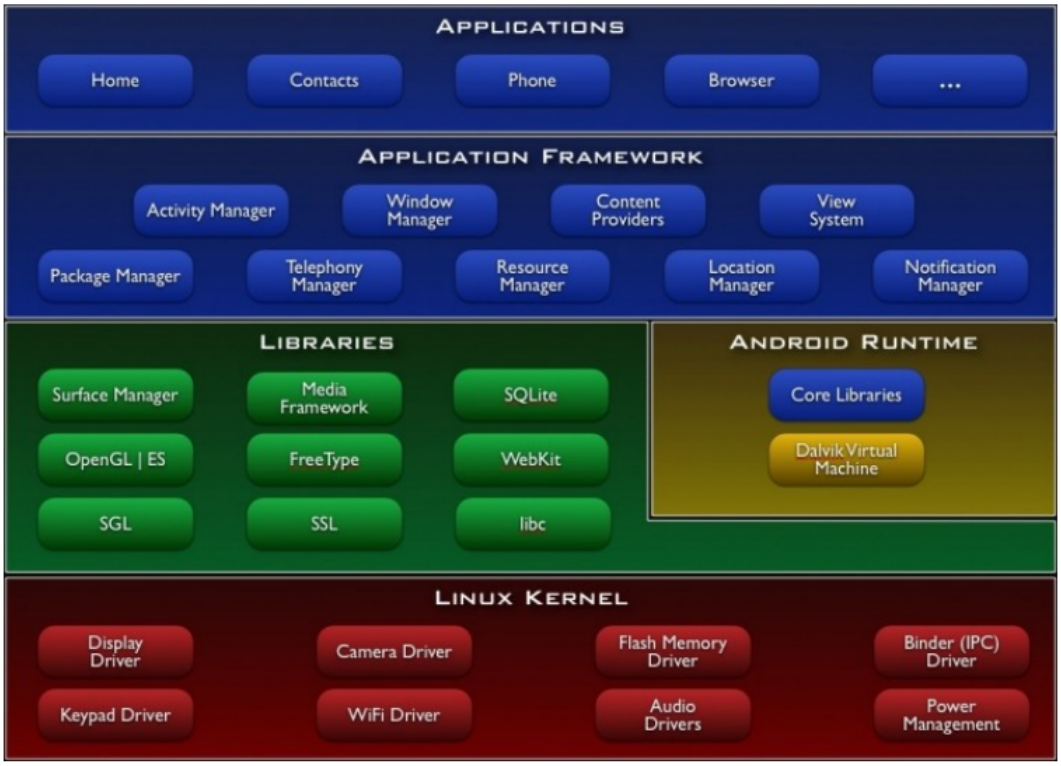
`compileSdkVersion` per fill-in manfest e `buildToolsVersion` per scaricare tools se non presenti.
`Lint` analisi statica di codice per warnings e errori sintattici della scrittura del codice.

1.3 Architettura Android Studio/Gradle

IntelliJ con plugin: android plugin. android designer, android gradle adapter.
Si appoggia all'android SDK e a Gradle (tool separato, con plugin android e anch'esso collegato all'SDK)
Inoltre c'è il progetto, con `.properties` con config per l'ambiente di sviluppo (come dove si trova il compilatore ecc.) e `build.gradle`.

Capitolo 2

Architettura Android



Studieremo a fondo le **applicazioni** e parte dell'**Android Runtime**. Le **libraries** sono largamente invisibili e il **linux kernel** è utile da sapere.

2.1 Struttura

Bottom – Up

Kernel Linux Alla base di tutto c'è il **kernel Linux standard, senza personalizzazioni**. Gli **adattamenti** per la parte telefonica sono **eseguiti tramite moduli del kernel**. Come display driver, driver per la tastiera keypad, driver camera, wifi, memoria flash, audio, driver binder (IPC) che cura Inter-Process Communication (diversamente da socket e FIFO, che non andavano bene per Android. **Su Android non ci sono solo file, ma oggetti con metodi**, e FIFO/socket adatte per trasferire flussi di byte. Il **binder fa comunicare processi in termini object-oriented**).
Per ultimo c'è il power management driver.
Per il resto è il kernel linux tanto conosciuto e amato: utenti, diritti, shell, librerie, thread e comandi.

Librerie Librerie `.so`, che fanno tantissime cose. Tra esse ci sono: surface manager (equivalente dei window system X o wayland), OpenGL ES per la grafica 3D, FreeType, SSL (HTTPS e Secure Socket Layer), WebKit, SQLite, libc (scanf, strlen...).

Android Runtime In aggiunta alle librerie, c'è anche la **Macchina Virtuale che esegue il codice delle applicazioni Android** (**Dalvik/ART**), insieme alle core libraries (garbage collector, heap, memoria...)

Separazione tra mondo Java e mondo del codice eseguibile ARM Da qui in poi c'è il linguaggio Java, fin'ora il linguaggio (del kernel linux) è il C.

Application Framework S.O. rappresentato da oggetti nello heap. Librerie: package manager, telefony manager, activity manager, window manager, location manager (GPS), notification manager

Applicazioni Tra cui servizi interni: home, contatti, telefono, browser... Firmate a chiave asimmetrica. Platform key per applicazioni che usano funzioni critiche, chiave che firma il kernel.

2.2 Dalvik & ART

Dalvik La stragrande maggioranza delle applicazioni gira su una macchina virtuale: **Dalvik**. Funziona in maniera analoga alla JVM con importanti differenze:

Basata su **registri** e non su stack

Set di istruzioni ottimizzato per risparmiare memoria e aumentare la velocità d'esecuzione

Formato dei file eseguibili ottimizzato per risparmiare memoria

Eseguibile da più processi con una sola istanza: tutto **codice rientrante** e sharing del codice di Dalvik via `mmap()`.

Non sotto il controllo di Oracle (storica causa legale)

Due meccanismi Fino ad Andorid 4.3 l'unico meccanismo di esecuzione era Dalvik. Durante Android 4.4 si è aggiunta l'opzione per eseguire su ART, con Dalvik come opzione default.
Da Android 5 in poi si esegue su ART.

Android Runtime ART ha delle differenze importanti rispetto ha Dalvik:

ART **pre-compila a install-time**, non interpreta
Questo rende l'installazione più lenta, ma l'**esecuzione più veloce**

Processo largamente **invisibile** a programmatore e utente

Però utilizzano lo stesso bytecode, producendo però **codice nativo** invece che bytecode: ulteriore assicurazione contro le cause di Oracle

Entrambi rilevanti Dalvik e ART sono entrambi rilevanti perché

Dalvik perché è il target della toolchain di compilazione del 99% delle app

ART perché Google da tempo sviluppa e supporta soltanto questa:

- Più veloce in esecuzione
- Miglior gestione della garbage collection
- Maggiore integrazione con profiling e debugging
- Minor consumo di energia

Fase	Dalvik	ART
Compile-Time	**javac**: `.java` → `.class` **dx**: `.class` → `.dex`	**javac**: `.java` → `.class` **dx**: `.class` → `.dex`
Install-Time	**dexopt**: `.dex` → `.odex`	**dex2oat**: `.dex` → `ELF`
Run-Time	**libdvm.so**: `.odex` → run Interpretato + JIT	**libart.so**: `ELF` → run Esecuzione nativa con un po' di runtime

JIT = **Just-In-Time** compilation, compila pezzi di codice che interpreta più volte

Esecuzione

Ogni app viene eseguita dal kernel Linux

> **In un processo separato**, che esegue Dalvik che esegue il bytecode dell'app: **controllo i permessi d'accesso alle risorse logiche fatto dalla VM** (permessi concessi dall'utente)
>
> **Con uno user ID distinto**: tutti i file creati dall'applicazione appartengono al proprio user ID, quindi altre applicazioni **non possono accedere** alla sua directory né **leggere i suoi file**. Applicazioni "*amiche*" (specificato nel `Manifest`) possono condividere processo e user ID, ma devono avere la stessa firma.
> Il **controllo dei diritti di accesso alle risorse fisiche è fatto dal kernel** (i diritti **non** sono controllati dall'utente)
> Riferisce la singola installazione sulla singola macchina.

Disaccoppiamento fra processo e programma. Il processo è un flusso di esecuzione dell'applicazione, ma **essa esiste indipendentemente dal processo**. Lo stesso processo può eseguire applicazioni diverse, la stessa applicazione può essere eseguita da processi diversi in momenti diversi.

Risultato Si ha un notevole grado di **separazione** e **isolamento** delle app, anche se ci possono essere sempre bug non scoperti all'interno del kernel Linux.
Android è un sistema piuttosto **sicuro agli attacchi**, eccezione fatta per quelli di ingegneria sociale: un utente può concedere dei permessi ad un'app malevola che li usa per scopi diversi da quelli pubblicizzati.

Utente L'utente non fa niente, se non tramite le app: non ha un user ID, non è proprietario di nessun file né titolare di nessun processo, non ha nemmeno delle credenziali di login.

Capitolo 3

Struttura di un'App Android

Diverse forme Durante la sua vita un'applicazione assume diverse forme:

Sviluppo, layout su disco sottoforma di **progetto**

Deployment, formato dei file `.apk`

Esecuzione, struttura in **memoria**

Le varie forme sono legate da tre processi:

Build: sorgente → `.apk`

Deploy: `.apk` su un canale di distribuzione (market...) → `.apk` sul device

Run: `.apk` → processo **in memoria**

3.1 Progetto

Un'applicazione in sviluppo è un **progetto**: lo scheletro viene creato automaticamente dal wizard di creazione di un nuovo progetto e **solo alcune directory sono interessanti per lo sviluppatore** mentre altre sono generate automaticamente.

3.1.1 Android Studio

manifests

`AndroidManifest.xml`: metadati dell'applicazione

java: sorgenti (`.java`, `.kt` e unit test associati)

assets: file arbitrari aggiunti all'`.apk`

bin: risultato della compilazione (risorse, `.dex`, `.apk`...)

res: risorse note al runtime Android

animator: animazioni basate su proprietà
anim: animazioni basate sull'intercalazione
color: colori
drawable: immagini raster o vettoriali
layout: descrizioni dei layout della UI
menu: menù usati dall'app
raw: file arbitrari, alternativa ad **assets**
values: costanti (stringhe, interi, array...)
xml: file XML arbitrari, incluse le configurazioni

libs: librerie native custom

Gradle Scripts: configurazioni di build

Altri come `*.properties`, `*.cfg`, `*.xml`...: configurazioni varie

3.1.2 Struttura di un progetto

La struttura vista è quella tipica di un'applicazione. Esistono altre due forme di progetto Android:

Libreria, contenente componenti destinati ad essere usati da altre app. In questo modo i membri di una famiglia di app correlate possono condividere componenti. Ogni libreria può essere usata da varie app e ogni app può usare varie libreria.

Progetto test, contenente codice usato per fare il testing di un'altra app.

3.2 APK

3.2.1 Contenuti di un APK

Il **build** di un'app Android produce un file in formato `.apk`, che non è altro che una **specializzazione di uno** `.jar` (che a sua volta è una specializzazione di uno `.zip`). Contenuti:

`resources.arsc`, file binario contenente la tabella che mappa ID a risorse

`classes.dex`: tutti i `.class` dell'app, convertiti in DEX e unificati in un unico file

`AndroidManifest.xml`, il manifesto

res/*, file delle risorse

META-INF/*, contenente i certificati pubblici (chiavi), solo per le app firmate

`META-INF/MANIFEST.MF` che contiene **informazioni di versionamento** e un **checksum di ciascun file**

`META-INF/CERT.*`, certificati RSA
Per il deploy è **necessario creare un proprio certificato**, quello di debug è generato da ADT

Un `.apk` è dunque un **archivio contenente tutti i componenti di un app**:

Auto-descrittivo grazie ai manifesti

Compatto grazie alla compressione

Affidabile grazie alla firma digitale: non è possibile aprire un `.apk`, sostituire alcuni componenti e re-impacchettarlo perché **la firma sarebbe invalidata**

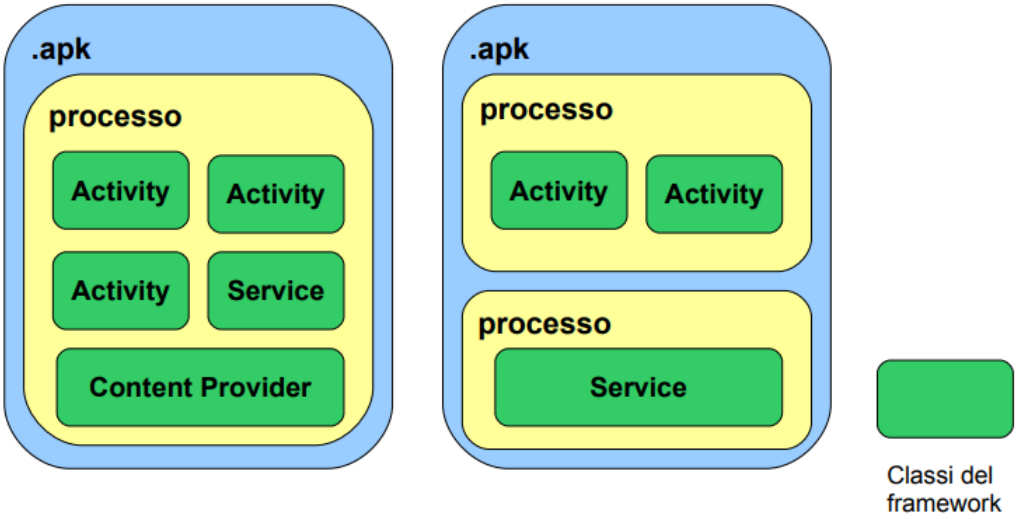
Facilmente **distribuibile** perché è un file unico

Facilmente **installabile**, niente wizard di installazione
In **/sys/app** se preinstallata o in **/data/app** se installata dall'utente

3.3 Memoria

3.3.1 Struttura in memoria

Una volta caricata in memoria un'app si distingue in **componenti**. Solitamente 1 app = 1 processo = 1 VM = 1 thread, ma possibili variazioni: multithread, app amiche sullo stesso processo...
Il flusso di lavoro dell'utente (**task**) è spesso fatto di componenti appartenenti ad app diverse, su processi diversi. Android **incoraggia la condivisione sicura fra app** sia di dati che di componenti = funzionalità.



L'app a sinistra, per esempio, è formata da 5 componenti. I componenti che vedremo durante il corso possono essere dei seguenti **tipi**:

Activity: una "schermata" dell'app

Content Provider: un fornitore di dati condivisi

Service: un fornitore di servizi condivisi (senza UI)

Un **task**, cioè quello che vede l'utente, può essere formato da un'activity dell'app a sinistra e le due activity dell'app a destra. C'è quindi una **importante distinzione** tra le visioni di programmatori e utenti:

un programmatore percepisce come "app" un `.apk`

un utente percepisce come "app" un **task**.

Quindi per garantire una UX gradevole occorre che l'integrazione sia **seamless**: uso di stili, temi e icone simili (**style guide**), **consistenza fra le app** più importante della consistenza dentro le app o fra piattaforme

3.4 Navigazione

Navigazione: utente passa da un contesto all'altro. Navigazione gerarchica, posso dal task leggere una mail e tornare indietro o andare a leggere il resto delle mail.
Originariamente solo tasto back, stack push e pop. Per lungo tempo ha asservito agli scopi, ispirato al paradgima navigaz web. Poi non era più sufficiente. Porta a sistemi nato per esser epiù logico ma confonde di più. Attualmente almeno due diverse: tasto back risale stack attività dove sono stato (Schermata in cui ero prima) e tasto up torna a livello logico superiore di dove sono (non è detto che ci sia già stato, solitamente implementato come icona nella actionbar).
Caso tipico è navigazione con pattern master-detail. Classico pattern di progettazione app. Pattern: soluzione adattabile per problemi frequenti. Nel master detail problema generale è: ho lista di cose e poi ho dettagli su quelle cose. Elenco mail ma posso leggerle, elenco chat ma posso entrare nella chat.

Capitolo 4

Risorse

4.1 Accesso alle Risorse

Ho risorse nella cartella res organizzate. Produce classe R.java che dà nomi usabili tipati. Efficienza perché uso interi per riferire oggetti di vari tipi (layout, stringhe...) che non vengono creati sullo heap. Al tempo stesso controllo staticamente tipi a seconda di R.layout, R.string... Così alloco gli oggetti solamente quando servono, all'ultimo momento. Accesso alle risorse metodi della classe `Context`, superclasse dei `Component`. Ogni pezzo dell'app è un context, da un thread, in una classe, in un'app, eseguita forse da un processo. `Context.getResources().get...(int id)`. Ci sono anche metodi per accedere alle risorse `raw` (risorse di tipo non interpretabile dal sistema): `InputStream openRawResource(int id)`
`AssetManager getAssets()` l'asset manager sola letture legge file: list lista file, assetfiledescriptor: file da usare in contesto più posix, magari per passarlo a metodo nativo scritto in C o a decoder assembler e inputstream: file da leggere ancora nel mondo java wrapper decode...

4.1.1 Risorse alternative

Per un certo numero di condizioni dell'ambiente si può indicare quali risorsi usare in che contesto: lingua, dimensioni schermo...
ID risorse è identificatore logico. Alcuni esempi...
Questo è fatto tramite qualificatori che descrivono aspetti dell'ambiente, si affiancano alle sottodirectory di res: `res/tipo-qualificatori`. Es `res/drawable-ja` icone da usare su dispositivi in lingua giapponese. `drawable-ldpi`, `drawable-hdpi`
...carellata qualificatori...Runtime processo di scelta. processo di scarto rimango con più opzioni scelgo quella per priorità, rimango con una ad un certo punto scelgo quella, rimango con 0. Se nessuna matcha tutti validi, se ne matcha una o più scarto i non validi. Inoltre c'è directory di default.

Capitolo 5

Componenti di un'app Android

5.1 Ciclo di Vita

Tipicamente un'app Android esegue così

Il launcher (che è esso stesso un'Activity) avvia la prima Activity dell'app **inviandole un'Intent** che indica l'intenzione di lanciarla. L'activity chiama `setLayout()` per impostare la propria UI

Il sistema chiama certi metodi dell'Activity (**callback**) in risposta alle azioni dell'utente
A seconda dei casi, questi metodi potranno

Lanciare altre Activity (inviando opportuni Intent), sia dell'app che di altre app

Inviare Intent ad Activity già in esecuzione o in broadcast a tutti gli interessati

Interagire con Services in background

Recuperare o salvare dati tramite un Content Provider

Terminare l'Activity, tornando alla precedente

5.2 Componenti

Cooperazione Abbiamo visto che le applicazioni Android non sono un blocco monolitico ma un insieme di componenti cooperanti:

Activity
Compone un'attività atomica dell'utente, concretizzata in una "**schermata**". Può essere **composta** da vari **Fragment**

Service
Attività di sistema o dell'app **invisibile all'utente**, eseguita in background. **Non interagisce con l'utente** ma **può interagire con le applicazioni** in vari modi.

Content Provider
Un componente che **pubblica "contenuti"**, con un'interfaccia programmatica che viene utilizzata da altre app.

Broadcast Receiver
"**Ascolta**" i messaggi **globali**. Quando riceve un messaggio di suo interesse, esegue del codice specifico (ad esempio, lancia un'attività).

5.2.1 Intent

I vari componenti **dialogano attraverso un sistema di messaggistica** che è alla base di Android

Intent Un Intent è un **messaggio** che **esprime un'intenzione di un utente o un'app affinché avvenga qualcosa**.
Ci sono numerosissimi intent di sistema, ma ogni app può definirne altri. Una **parte della struttura** del messaggio **è fissata**, ma possono **includere dati "extra" a piacere**.
Gli intent possono essere **indirizzati ad uno specifico componente** oppure **emessi in broadcast**.
Ogni app definisce un **filtro** che **dichiara a quali intent è interessata** e può eventualmente rispondere.

Per **avviare un'Activity** ci sono due modi

Explicit Intent: possiamo creare un Intent che **chiede un'Activity**.
Se l'Activity in questione non è in esecuzione allora viene lanciata.
Se è già in esecuzione viene "svegliata".
In entrambi i casi, viene recapitato l'Intent e l'Activity destinataria diventa "attiva".

Implicit Intent: possiamo creare un Intent che **chiede una funzione**.
Il **sistema cerca quale Activity può rispondere**. Se ne trova una, la attiva.
Se ne trova più di una, chiede all'utente quale lanciare.
Se non ne trova nessuna, fallisce.

Esempio di Implicit Intent `ACTION_SEND`. L'app crea un Intent implicito con `ACTION_SEND`, ci mette dentro i dati da inviare e lo affida al sistema che cercherà le app installate che supportano `ACTION_SEND` e chiederà all'utente quale usare.

5.2.2 Contenuti di un Intent

Action: quale azione si vuole ottenere (`String`)

Data: su quale dato operare (`URI`)

Type: tipo MIME di Data (`String`)
Se non è fornito viene ricavato da Data, se è fornito fa override su quello di Data.

Component: componente a cui è indirizzato il messaggio.
Solo per Intent espliciti

Extras: un `Bundle` (mappa chiave → valore) di ulteriori campi

Flag: ...

5.2.3 Intent Filter

5.2.4 Activity Stack

Utente interagisce solo con la top ma può vedere le altre parti se la top ha parti trasparenti.
Ci sono tanti stack. In un dato momento sono in uno stack. Se scelgo di cambiare schermata vado sull'activity precedente a questa in ordine cronologico sempre sul suo stack.
Politica con cui vengono avviate activity... (p.36)
Comportamento dipende da due flag: uno deciso da chi manda l'activity e altri flag decisi dall'activity.
`launchMode` (manifest)

`Standard` crea sempre nuova istanza

SingleTop riavvia istanza se activity top altrimenti nuova istanza che diventa top

SingleTask Una sola istanza, se ce n'è una in qualche stack viene riavviata quella altrimenti nuovo stack

SingleInstance Si crea nuobo task che non consente altre activity (es.: Home)

Con i `FLAG_ACTIVITY_*` ci sono ulteriori opzioni per essere più specifici

Disattivare l'animazione di transizione

Lanciare senza che appaia nella history

Decidere se spostare l'activity in cima allo stack o lasciata al suo posto

...

Flag più comuni (intent)

new_task

5.2.5 Lanciare activity

Mai usare `on<qualcosa>` nel codice scritto da noi, sono chiamate fatte dal sistema.

5.2.6 Layout

Il layout è impostato con `setLayout()`, ma può cambiare dinamicamente: alterando il layout o manipolando i fragment che compongono l'activity.
Nel secondo caso i cambiamenti ai fragment vengono messi nel back stack.

5.2.7 Terminazione

Activity può finire *suicidandosi* con il metodo `finish()`, con `setResult()` chiamato prima per pubblicare l'eventuale risultato.
Si può chiudere un'altra activity chiamata con `startActivityForResult(intent, codr)` tramite `finishActivity(codr)`.
Il sistema può uccidere activity in qualunque momento per liberare risorse o quando cambia la configurazione (landscape, cambio lingua, modalità notte...).
Il sistema avvisa prima di uccidere, ha un protocollo il sistema basato su tre classi di priorità (dei componenti)

Priorità Critica, mai uccisi. Activity in cima allo stack corrente (quindi, in uso). Di solito una sola, ma in caso di Picture-In-Picture o schermo diviso possono essere di più.

Priorità Alta, uccisi alla disperata. Activity visibili attraverso le aree trasparenti dell'activity top.

Priorità Bassa, uccisi in ordine LRU se serve. Activity non visibili in fondo allo stack corrente o in altri stack.

Quindi bisogna essere pronti a salvare stato e ripristinarlo, con metodi apposta.

5.3 Ciclo di Vita di un'Activity

Fase di avvio

`onCreate`: codice in fase di creazione, inizializza stato

`onStart`: codice subito prima di essere visibile, UI pronta

`onResume`: stai per passare da potenzialmente visibile a responsive, stato in cui prendi input da utente

Qua l'activity è in top stack, visibile e interagente con l'utente.

`onPause`: stai per diventare non interagente con l'utente, ad es.: altra activity và sopra di te (siamo già a rischio, bisogna salvare dati!)

`onStop`: non sono più visibile (posso interrompere animazioni, ad esempio), non occorre tenere UI aggiornata

`onRestart`: torni visibile

`onDestroy`: l'activity non esiste più, abbiamo finito o stanno per ucciderci (salvare dati!)

Se il processo è ucciso

Ritorno alla `onCreate`

Da A a B

A pause

B create

B start

B resume

A stop, se B non ha parti trasparenti

L'implementazione **deve** chiamare il metodo corrispondente della superclasse

```
@Override
protected void onStart() {
super.OnStart();
//codice
}
```

Perché super fa tanto lavoro che deve per forza fare anche se viene fatto Override

5.3.1 Salvare lo stato

Callback `onSaveInstanceState()` chiamato dal sistema quando sta chiedendo di salvare istanza. Prima di `onStop()` e `onPause()`. Se vi chiama, è arrivato il momento di salvare, ma è possibile che non venga chiamato (se sistema sa che non può tornare a chiamarvi, come quando utente chiude task da lista task recenti).
Activity is running → altra attività viene in sovraimpressione quindi `onSaveIstanceState()` → activity non più visibile → ... → resume dell'activity. Non viene ripristinato stato perché processo non è stato ucciso

Activity is running → altra attività viene in sovraimpressione quindi `onSaveIstanceState()` → activity non più visibile → ... → processo ucciso → utente torna all'activity (che non è stata distrutta) → sistema **ricrea l'activity** → `onCreate()` ripristinando stato salvato

5.3.2 Bundle

Bundle: coppie chiave-valore.
Metodo `onSaveInstanceState()` dovrebbe salvare tutto quello che può servire: `putString`, `putInt`... anche `putBundle`!
Activity distrutta ma bundle sopravvive, verrà poi passato a `onCreate()`, e anche a `onRestoreInstanceState()` chiamata dopo `onStart()`.

5.3.3 Default

Classe activity ha implementazione di default di salvataggio e ripristino stato: scorre il suo layout e salva nel bundle tutte le view chiamando il loro `onSaveInstanceState()`, ma solo quelle che hanno un `id`. Se l'activity ha solo UI basta così, ma conviene praticamente sempre chiamare `super.onSaveInstanceState()` anche se non si devono salvare ulteriori pezzi di stato (non-UI).
Questo consente di ritrovare il campo riempito precedentemente se si torna all'activity.

Capitolo 6

Interfaccia Utente

6.1 Layout

UI Android è un albero con foglie di classe View e nodi intermedi di classe ViewGroup.
Ogni View è una classe Java con nome uguale al tag XML relativo.
Quando si verifica un evento significativo sulla View viene chiamato un **handler** della View. Il layout è definito in file XML, con oggetti `Inflater` che si occupano di parsare gli XML e istanziare le classi Java corrispondenti all'albero. Non regola la posizione, che è gestita da algoritmi che stanno in `ViewGroup` particolari chiamati `Layout`.
Una View che gestisce input è tipicamente chiamata Widget, e un contenitore di Widget è detto Widget Host (come la Home).
Il LayoutManager si occupa del posizionamento dei propri figli. Il layout avviene tramite una **negoziazione** tra posizionatore e posizionati (padre e figli). Il figlio non può sapere a priori dove viene agganciato, lo saprà solo a runtime. Ci sono tante strategie di negoziazione, quindi tanti layout manager. Tra i più comuni, troviamo:

`AbsoluteLayout`
Coordinate assolute `(x, y)`

`LinearLayout`
Serie verticale o orizzontale di componenti

`RelativeLayout`
Posizione di ogni componente relativa agli altri o al container

`GridLayout`
Griglia di celle di dimensione variabili, ma allineate. Componenti anche a cavallo di più celle

`FrameLayout`
Componenti uno sull'altro, una sorta di trasparenza: l'ultimo aggiunto sta in cima e tutti grandi quanto il container. Solitamente usato con un solo componente.

`TableLayout`
Alternativa a `GridLayout`. Ha come figli singole View (riga intera) o oggetti `TableRow` (che hanno come figli le view delle singole celle)

Sempre possibile comporre layout o scrivere i propri (vedremo come).
Ci sono LayoutManager più specializzati:

`DrawerLayout`
Per i menu a scorrimento: un figlio è l'interfaccia principale, l'altro è il drawer

`SwypeRefreshLayout`
Se l'interfaccia è trascinata verso il basso spunta un'icona e si richiede il refresh

...

Tante View sono *anche* Layout

`Toolbar`

`TvView`

`Webview`

`CalendarView`

`Gallery`

Circa una cinquantina...

Su Android Studio vengono proposti i layout più comuni, con editor grafico e testuale per scrivere l'interfaccia. Sempre possibile editare direttamente l'XML.

6.2 Interazione

A run-time, esiste un albero di oggetti Java che è stato creato a partire dall'albero XML del layout. Gli oggetti possono ricevere input dall'utente (si interfacciano col sistema touch).
Quando si verifica un evento significativo, viene chiamato un **handler**: l'Activity può registrare propri handler.
In Java, sono inner interfaces dentro la classe View. Ogni interfaccia definisce un metodo on...Listener()

6.2.1 Menu

Logici Su Android il menu (o meglio: le opzioni) **non è un componente grafico ma uno logico**.
In altre parole, l'app dichiara quali scelte devono essere disponibili all'utente, ma il **sistema decide autonomamente come presentarle**:

in base alla versione del S.O., allo spazio disponibile su schermo, alla presenza di una tastiera o meno, se siamo su TV, automobile e altro ancora...

in base al fatto che siano opzioni o azioni, a quanto è lunga l'etichetta, se c'è icona...

in base alla priorità specificata dal programmatore, anche se è comunque interpretata dal S.O.

Non convenzionale Android usa un sistema non convenzionale per i menu: niente liste gerarchiche con etichette.
L'implementazione tradizionale è su tre stadi:

Menu primario di 6 caselle, o 6 opzioni o 5 opzioni + "*altro*"

...

Sistema decide **autonomamente** come, dove e quante voci mostrare.
Da **Honeycomb** le voci del menu sono spostate nella actionbar. Per **retrocompatibilità**, le app senza actionbar su dispositivi senza tasto menu fisico hanno un pulsante aggiunto ai softkeys (tre puntini).

Creazione di un menu

...

Ciclo di Vita di un Menu

Non ha senso allocare in memoria oggetti del menù finché non è aperto dall'utente. Nostra activity fornisce callback per crearlo (`onCreateOptionsMenu`), prepararlo (`onPrepareOptionsMenu`) e reagire alle azioni (`onOptionsItemSelected`).
Da android 3.0 onCreate è chiamata una volta sola alla creazione dell'activity (alcune voci finiscono sull'action bar).
Fino a 3.0 era chiamata una sola volta, sempre, ma all'apertura del menu.
OnPrepare viene chiamta ad ogni apertura del menu prima di 3.0, da 3.0 solo se prima si chiama `invalidateOptionsMenu`.
La onPrepare è dove si possono apportare modifiche al menù: abilitare/disabilitare voci...
Deve restituire true se è disponibile, o false se non c'è menù da mostrare.
Evitare eccessiva variabilità del menù, altrimenti l'utente si perde. Usare menù contestuali, altri controlli UI...
Ricordare di invalidate se si vuole eseguire un'altra volda prepare.

Posso associare ad un `menuItem` un intent, che viene lanciato quando l'item viene toccato. A chi arriva arriva.
Si può configurare il `menuItem` per chiamare un listener.

6.2.2 Menu Contestuali

Equivalente del tasto destro. ConextMenu associati ad una particolare view. Dipendono dalla particolare view, mentre gli optionsMenu dipendono dall'activity essendo più globali.

6.2.3 Context Actions

Alternativa ai context menu, si sovrappone visivamente all'actionbar dell'activity, ma è un oggetto separato.
Si implementa con l'interfaccia `ActionMode.Callback`, con metodi analoghi a quelli per gli `OptionsMenu`.
`startActionMode()` per aprire l'action bar contestuale.

6.3 Scrivere le proprie View

Elementi UI sono sottoclassi di view: view, viewgroup, layput, widget. organizzazione spaziale è scritta in un file xml che crea il layout, o comunque dal layoumanager.
Per creare un widget è sufficiente estendere view o una delle sottoclassi. Override di due metodi pressoché essenziali:
`onMeasure(in widthSpec, int heightSpec)` chiamato per negoziare le dimensioni: parametri passati sono i requisiti (dimensioni massime che può assumere) che il contenitore vuole inestigare. il metodo onmeasure **deve** fornire risposta chiamando il metodo `setMeasuredDimension(int width, int height)`
`onDraw()` chiamato per disegnare l'effettiva UI
Altri per gestire l'input: `onKeyDown()`, `onTouchEvent()`...

`widthSpec` **e** `heightSpec` Hanno due componenti: modo e dimensione. 2 bit per il modo, 30 bit per la dimensione. Per estrarre valori, `getMode()` e `getSize()` di `View.MeasureSpec`.
Il modo può essere `UNSPECIFIED` no vincolo, `EXACTLY` contenitore impone esattamente la dimensione data o `AT_MOST` contenitore impone un massimo

Quindi onmeasure guarda vincoli e risponde specificando la propria dimensione preferita. Possiamo non rispettare i vincoli, ma poi il disegno è fatto dal contenitore, quindi clipperà. *Il sistema vince sempre.*
Solitamente i minimi sono suggeriti, indicati dai metodi `getSuggestedMinimumHeight()` e `getSuggestedMinimumWidth()`, utili ad esempio per evitare di fare una view da toccare più piccola del dito.

`onDraw(Canvas)` Il Canvas è una superficie di disegno, molto più sofisticata di una bitmap: insieme di operazioni con cui disegnare. Per disegnare sul canvas si usa la Paint.
Canvas include anche tutti i metodi con cui disegnare: primitive di disegno (linee, cerchi, testo...), primitive per il clipping (limitare l'area e ignorare disegno fuori di essa) e matrici di trasformazione
Canvas permette di indicare disegno, per poi realizzarlo successivamente

Pipeline di rendering

Rendering di path, trasformazione secondo PathEffect (arrotondamento, tratteggio...)

Rasterizzazione: alpha-channel, aliasing, filling, blurring...)

Shading: trasformazione colori, gamma...

Trasferimento: disegno parziale combinato con precedenti contenuti della bitmap con vari operatori

Diagramma di flusso

Metodi di `Canvas`

Clipping: `clipPath()`, `clipRect()`, `clipRegio()`

Trasformazioni affini: rotazione, scala, traslazione...

Primitive di disegno: `drawBitmap()`, `drawLine()`...

Informative: `getWidth()`, `isOpaque()`, `getDensity()`...

Di ogni metodo ci sono tantissime varianti.

`Paint` Specifica completa di come disegnare una primitiva di Canvas: colore, dithering (se voglio colore non disponibile es: schermo e-ink, lo disegno in qualche modo con quello che ho), font, alpha, tratteggio, dimensione delle linee...

6.3.1 Costruttori

Bisogna implementarli tutti e tre

`View(Context c)`

`View(Context c, AttributSet attr)` con attr codifica di ciò che c'è nel file xml

`View(Context c, AttributeSet attr, int stile)` applica uno **stile**

Non c'è `View()`: non è possibile istanziare View senza contesto.

Una view ha delle proprietà grafiche e di contenuto. Serve un mezzo per ottenere il refresh della vista quando cambia la proprietà: `invalidate()`. Alla prima occasione il sistema richiama `onDraw()` e ridisegna la view.

6.4 ListView & RecyclerView

ListView Uno dei componenti più comuni. Ogni elemento è a sua volta una View: la ListView è una lista di elementi. I dati delle viste figlie sono dati arbitrari, quali sono lo decide l'adapter.
Possono essere statiche, con informazioni caricate per esempio dalla cartella `values`.

6.5 WebView

Controllata da WebChromeView, WebSettings e ...
Esegue nativamente codice js, possibile binding fra oggetti java dell'app e oggetti js della pagina (`addJavascriptInterface(ogg`
`nome)`. L'oggetto Java diventa accessibile agli script js della pagina con il nome dato. Tecnica rischiosa in termini di sicurezza.

Usate per fare app cross-platform basate su tecnologie web: HTML5, CSS3, JS, Jquery, Bootstrap, node.js...
La si esegue dentro una webview arricchita: l'ambiente aggiunge una serie di oggetti predefiniti scritti in Java che fanno da "ponte", così è possibiel chiamare da javascript i metodi delle varie classi di android non emulabili da HTML5.
Tutta la grafica è renderizzata sul DOM.

Capitolo 7

Kotlin

Preferenza per: modernità, comodità e diminuire ulteriormente dipendenza da tecnologie proprietarie (Java)
`val` da errore se compilatore trova flusso d'esecuzione possibile con due assegnamenti.

Capitolo 8

Esecuzione Concorrente

8.1 Sistema e Callback

Come visto, le applicazioni si limitano a definire callback: `onPause()`, `onDraw()`...
C'è un thread, tipicamente definito "thread della UI", **principale dell'app**.
Due regole auree:

! **Mai usare il thread UI per le operazioni lunghe**
Tipicamente bloccanti come accesso a file system o alla rete, o computazionalmente pesanti, ma in generale tutte le operazioni che richiedono più di 100ms.

! **Mai usare un thread diverso dal thread UI per aggiornare la UI**

Come fare se serve un'operazione lunga che deve aggiornare la UI? Es: accesso a DB, alla rete, calcoli *pesanti*...
Creare nuovi thread aiuta per la prima regola, non per la seconda.

8.1.1 AsyncTask

Asincrono $\neq$ concorrente! Fai del lavoro, poi riprendi dopo senza avere illusione di essere l'unico con la CPU.

Caso comune Thread UI fa partire un task (lungo), che deve aggiornare la UI durante lo svolgimento e deve fornire il risultato alla UI alla fine.
Per questo particolare caso è **molto comodo usare la classe** (astratta e generica) **AsyncTask**.

`AsyncTask<A, P, R>` Prende argomenti del task di tipo A, progress report di tipo P e risultato di tipo R.

Chi esegue Prima di partire chiamata `void onPreExecute()`, al suo ritorno la `R doInBackground(A...)` che se chiama `void publishProgress()` chiamata la `void onProgressUpdate(P...)` e alla fine `void onPostExecute(R)`.
Su cancellazione anticipata `void onCancelled(R)`.
All'esterno ci sono i costruttori, `AsyncTask execute(A...)`, `cancel(boolean interrupt)`, `R get()` e `AsyncTask.status getStatus()`. Con `R get())` prendo il risultato restituito da `R doInBackground(A...)`, comodo per prendere il valore quando voglio io.
Il thread della UI esegue (devono essere veloci ma possibile interagire con UI:

`onPreExecute`

`onProgressUpdate`

`onCancelled`

Costruttori (ma possono essere eseguiti da altri thread)

`execute`

mentre un nuovo thread esegue (possono essere lenti ma no interazione con UI):

`doInBackground`

`publishProgress`

WeakReference Normalmente riferenza a oggetto impedisce a garbage collector di eliminarlo. Il nostro bitmapdownloadertask avrebbe riferimento a imageview, se non avessi usato una weakreference. Però imageview avrebbe riferimento al proprio layout, che ha riferimenti all'albero, che è radicato in un context che ha riferimenti a tantissime cose. Quindi sarebbe praticamente impossibile liberare la memoria. Si comporta come un riferimento ma non impedisce al gc di disallocare la memoria (sempre che non ci siano altri riferimenti in giro)
In caso di disallocamento, il `get()` della WeakReference ritorna null.

Cursor Generalmente sono lenti e bloccanti (quindi no thread UI), o thread a parte oppure Loader.

Loader Classe **astratta** specifica **protocolo generico** per caricamente asincrono di dati.
Si tratta di classi di utilità, **componenti attivi**: a Loader attivo, quando i dati sottostanti vengono aggiornati, il Loader deve trasferire aggiornamenti al suo utente. Il Loader stesso non specifica come fare.
Ogni activity ha associato un LoaderManager, che gestisce i Loader di quell'activity (analogamente ai FragmentManager). Uno solo per activity, che gestisce un numero qualunque di loader. Offre initloader (crea loader), getloader, restartloader, destroyloader.
CursorLoader è una sottoclasse di Loader: interroga un ContentResolver in background.
Il pattern diventa da aprifile a inizia ad aprilo e ho finito di aprirlo.
StarLoading, *OnStartLoading*, StopLoading, *OnStopLoading*... (*corsivo* = astratte, da implementare).
Attenzione: deprecati da Android 9

Download HTTP con DownloadManager Succede tantissime volte: mappe, api...
Come si usa, servizio di sistema: getSystemService(DOWNLOAD_SERVICE). Posso **accodare** una richiesta di scaricare qualcosa (**long id = downloadmanager.equeue(request);**. Al termine si viene avvisati con un broadcast (ACTION_DOWNLOAD_COMPLETE), quindi bisognerà registrare un BroadcastReceiver.
`request` deve essere un'istanza di `DownloadManager.Request`, incapsula una richiesta di download e contiene tutte le info necessarie fra cui

URI (obbligatoria)

Dove metterla (default spazio shared)

Se visualizzare notifica o no durante download

Titolo e descrizione per la notifica (Se ci sono)

Restrizione sulla rete (Solo wifi, anche cellulare, anche roaming...)

AsyncPlayer Molte classi di sistema gestiscono multithreading completamente al proprio interno.
Una di queste è AyncPlayer: riproduce audio in background, il thread c'è ma non si vede.
Due metodi importanti: play e stop. Play prende la URI dell'audio, stop non ha parametri.

Pattern command Trasformo azione (tipicamente chiamata di metodo) e trasformo in struttura dati.
Command diventa un dato da fare in un momento qualsiasi, ad esempio mettendolo in una coda.

Capitolo 9

Intent

Alcuni **devono** essere dichiarati dinamicamente. Di solito, registrazione in onResume e deregistrazione in onPause.

Sticky Un intent sticky rimane vivo, se viene lanciato altro intent viene consegnato anche il precedente se i filter combaciano.
Questo significa che l'intent sticky non può essere eliminato dal garbage collector: infatti richiede permessi speciali.

sendBroadcast(Intent intent) per mandare un intent in broadcast (serve permesso). Ha senso se fornisco info che interessano ad altre app. Intent mandato con sendBroadcast a broadcastReceiver diverso da intent passato a startActivity.
sendBroadcast è **asincrona**.

Varianti

Permessi: possibile consegnare Intent solamente a app con determinato/i permessi

Serializzazione: possibile invio serializzato ed ordinato. Normalmente i broadcastreceiver sono eseguiti concorrentemente.

Abort: in risposta a sendOrderedBroadcast(), recever possono abortire il broadcast e restituire risultati

Risultati: per avere risultati si attiva un *folding* con un resultReceiver. Si usa variante di sendOrderedBroadcast: con resultReceiver proprio broadcaastreceiver, schduler se coda personalizzata e tre variabili "accumulatore" int initialCode, Strign initialData, Bundel initialExtras.
Perché così? L'idea è folding. chiamo il primo breceiver nella lista e passo la lista initial che la modifica, la prendo e la passo al secondo che la rimodifica... e così via. Quando passo all'ultimo, la tripla modificata la mando al resultReceiver.

Sticky L'essere sticky viene determinato da chi li lancia. sendStickyBroadcast(Intent intent)

Esempio per stato di carica IntentFilter con `Intent.ACTION_BATTERY_CHANGED`, e Intent battery = registerReceiver(null, filer), null perché così prendo il più recente intent sticky.
Poi status = battery.getIntentExtras(BatteryManager.EXTRA_STATUS, -1)...

Broadcast Locali Con LocalBroadcastManager con metodi analoghi.

Capitolo 10

AlarmManager

Servizio di sistema: `Context.ALARM_SERVICE`, restituisce AlarmManager.
Ipostare sveglie e ricevere Intent alla ricezione di una sveglia.
Anche possibile `setInexactRepeating`.
Usare alarm solamente quando si vuole essere svegliati, per compiti di temporizzazione meglio csotrutti come quelli nativi Java (Thread.sleep(), System.currentTimeMillis()...) oppure Handler con postDelayed.
Alarm scattano anchea telefono in sleep. Durante la onReceive di chi ha ricevuto il pendingIntent, il telefono viene svegliato e tenuto sveglio. Ma al ritorno della onReceive può tornare immediatamente in sleep. Se fa partire qualche attività asincrona, questa potrebbe fermarsi subito perché il telefono torna in sospensione.

Cambiamenti Fino alal 4.0 alarm sempre esatti, a meno che non fossero esplicitamente inesattiv. Da Android 4 in poi, per le app con arget 19 o successivo, allarmi sono **inesatti per default**. Il sistema li aggiusta per ridurre risvegli e ottimizzare batteria. Per specificare l'esattezza, c'è `setExact`.
Si può usare `setWindow` per dichiarare "quanto inesatto" sono disposto ad accettare, cioè quanto aggiustamento concedere al sistema. L'allarme scatta in qualunque punto della finestra concessa.
Da Android 7 in poi c'è la modalità **Doze**, può ritardare l'attivazione di allarmi. A schermo spento e device in movimento gli allarmi funzionano normalmente.
Possibile settare allarmi che superino il Doze (setAndAllowWhileIdle, setExactAndAllowWhileIdle): ovviamente peggiora la durata della batteria.

10.1 WorkManager

uan delle tante api jetpack per task asincroni deferrable coordinati e vincolati

Asincrono: eseguito in qualche momento del futuro

Deferrabile: può essere rimandato senza danno

Coordinati: possono esprimenere strutture (task in seq, parall...) farm, pipeline...e passare mappe chiavi/-valori fra task

Vincolati: per ogni task, si possono specific condizioni che devono essere verificate per l'esecuz.

Un task da eseguire è un **istanza di Worker**, di cui definiremo una sottoclasse (perché è astratta) implementando **doWork()**. (*Worker sostituisce Runnable, doWork() sostituisce run()*).
doWork() restituisce un Result, ad esempio `return Return.success()`. doWork() è eseguito da un **thread non-UI**, gestito dal WorkManager.

WorkRequest Per richiedere l'esecuzione di un Worker (cioè task) definisco un'istanza di una sottoclasse di WorkRequest. *Richiesta di eseguire, non la cosa da eseguire.* Ha diverse sottoclassi, tra cui:

OneTimeWorkRequest: esegui task una volta sola

PeriodicWorkRequest: esegui il task ripetutamente

`WorkManager.getInstance().enqueue(myWorkRequest);` per mettere in coda una richiesta.
Ho costruttori di esecuzione per accodare richieste insieme da eseguire in parallelo o in sequenza: `WorkManager.getInstance().beg`
`2).then(3).then(4, 5).equeue()` eseguire 1, 2 in contemporanea, poi 3, poi 4, 5 in contemporanea.
Si può annullare l'esecuzione di un task con `WorkManager.cancelWorkById(wReq.getId());`

Vincoli Possibile specificare sotto quali vincoli un task è eseguibile. Vincoli rappresentati come istanze di Constraints, tipicamente vincoli esterni alla computazione. `Constraints.Builder().setRequires...(true)....build()`, e usare setConstraints(myConstr) per assegnarli alla WorkRequest.

Capitolo 11

Services

Service pezzo nostro di codice che non possiede interfaccia utente. ContentProvider BroadcastReceiver sono estremamente specializzati, mentre i Services sono più sofisticati: esecuzioni in background arbitrarie. "In background" nel senso che non hanno UI, quindi senza partecipare allo stack delle activity...
Il codice del service è eseguito sempre dal main thread, quello della UI.
Activity possono avviare uno o più Service, sempre nel proprio thread principale, perché rimangano in esecuzione indefinitamente o per lavorare e poi terminare.
Hanno cicli di vita distinti. Ci sono due modalità

Servire una sola richiesta: **started**
Il loro ciclo di vita è startService(), onCreate(), onStartCommand(), esecuzione, onDestroy(), spegnimento.

Stabilire connessione tra componente e service e rimane a disposizione per gestire un flusso di richieste: **bound**
Ha sempre una priorità nella scala delle uccisioni almeno uguale all'activity a cui è collegata.
Il loro ciclo di vita è bindService(), onCreate(), onBind(), esecuzione, onUnbind(), onDestroy(), spegnimento.

Come sempre, Android può uccidere un Service in qualsiasi momento.

Avviare un service started startService(new Intent(this, MyService.class));...

Terminare Il servizio stesso può terminare con `stopSelf()`, eventualmente con *id* di uscita.
Un altro componente può chiamare `stopService(Intent i)` passando l'intent che identifica il servizio.
S.O. può forzatamente chiudere i servizi.
S.O. non tiene traccia delle richieste mandate, usa l'high water mark (si segna solamente un intero che rappresenta più alto numero richieste ricevute fin'ora). Se si chiama `stopSelf(id)` con id inferiore al max allora ok, se è uguale al max considera il servizio come concluso. Si, è molto pericoloso.

Riavvio valore di ritorno da onstart è codice che dice come gestire riavvio

START_STICKY: servizio fermato appena possibile viene chiamato nuovamente onstartservice con null. tipicamente startservice, stopservice

START_NOT_STICKY: servizio fermato riavviato solo se ci sono chiamate a onStartService(). Tipicamente startservice, stopself

START_REDELIVERY_INTENT: se serivizio fermato verrè riavviato con l'intent originale.

Flag di avvio

0 è avvio

START_FLAG_REDELIVERY

START_FLAG_RETRY riavvio sticky dopo uccisione forzata

ComponentName Valore ritorno di startService(). Rappresenta il componente Service (è una classe generale, può anche rappresentare Activity, BroadcastReceiver, ContentProvider)
Se null Intento non consegnato
Altrimenti si può utilizzare per avere info: getClass, getPackageName...

Riassunto Service Started

Ottimizzazione batteria Alcune strutture
JobIntentService Come intent service ma come job periodici invece che come
startForegroundService() come starService()...

Bound Service Interazione limitata. Garantiscono disponibilità servizio per un periodo più lungo. chiamare metodi del service attraverso inter-process communication: in-process con un IBinder o cross-process con AIDL (Android Interface Description Language) (analogo a RMI e RPC)
Binding: service e componente legati in modo più stabile e comunicativo, connessione rimane fino ad unbound esplicito.

Iniziare un Binding bindService, servizio si identifica comunque con un Intent. Altro parametro: connection che controlla tempo di vita del binding. Su connessione lancia onServiceConnected(ComponentName name, IBinder binder) con binder interfaccia su cui chiamare i metodi
Su disconnessione onServiceDisconnected(ComponentName name)

Usare un servizio bound bindService(intent, conn, flag) → onBind(intent) del service e forse onCreate(). onBind restituisce binder che implementa l'interfaccia IBinder, passato alla onServiceConnected() di conn. Da qui in poii si usano i metodi del binder, alla fine si chiama unbindService(conn)

Rischio sicurezza se usiamo un intent implicito in bindService(): non possiamo sapere quale rispondere, ma qua è pieggio perché possiamo invocare metodi che il service che ha risposto non ha. Da Android 5.0 non si può chiamare con intent implicito (eccezione).

Mescolare bound e unbound Possibile che un Service offra sia interfaccia unbound e bound, perché onStartCommand() e onBind() sono separate. Tuttavia complica molto il ciclo di vita, meglio in generale scegliere uno stile e mantenerlo.

Capitolo 12

Sensori di sistema

Accesso ai sensori Android implementa sistema sensori del tutto generico. Esteso a tipi di sensori diversi, meccanismi simili ma **valori da interpretare**.
Servizio di sistema: **SensorManager**. Come per altri servizi, una volta ottenuto l'handle si possono usare i metodi.

Sensor Discovery getSensorList, restituisce lista di oggetti di classe Sensor. Chiede un qualificatore di tipo, con Sensor.TYPE_ALL chiedo tutti i sensori.
Ogni Sensor descrive un tipo di sensore: type, name, vendor, version, max range, min delay, resolution, power.
Si può avere più di un Sensor per tipo, uno è quello usato per default. Per ottenerlo getDefaultSensor(tipo).
Alcuni sono hw altri sw: integrando un sensore di accelerazione, si può ottenere un sensore di velocità, oppure con un filtro passa-basso si può determinare la forza di gravità. Allo sviluppatore non importa distinguerli (è anche una brutta idea tentare di farlo).

Sensori necessari Si può chiedere nel manifest che sul dispositivo ci sia un certo sensore: uses-feature.

Leggere sensori *Con calma e per favore*: si registra un listener indicando anche con che frequenza si vuole essere chiamati ma è il sistema a decidere quando chiamarti. Due callback:

onSensorChanged(): cambia valore letto (es: bussola e telefono viene ruotato)

onAccuracyChanged(): cambia l'accuratezza del sensore (es: passo da localizzazione GPS a quella radio *anche se GPS non è letto come sensore è solo un esempio*)

Molto sensato registrare listener nella onResume e de-registrarlo nella onPause(), così da leggere sensore solo quando activity è in primo piano.

Registrazione con registerListener(SensorEventListener l, Sensor sens, int rate): registra l per essere informato degli eventi relativi a senso con una frequenza di **circa** rate (espresso in microsecondi: ci sono costanti predefinite.
Il listener può essere creato come anonymous (new SensoEventListener dentro la chiamata) o implementarlo nell'Activity come al solito.

Accuracy indicazione realativa allo stato del sensore, **non è la risoluzione**.

SensorEvent

Cautele

+ precisione + batteria

+ frequenza + batteria

mai lasciare listener registrati per più del dovuto

considerare approcci adattivi: leggere poco e male quando non serve, aumento precisione e frequenza quando serve

non dedicarsi a operazioni lente nei metodi on. . . ()
Semmai, memorizzare in una struttura dati e processare in un thread (sincronizzazione!). Il thread di processing può avere il suo tempo di ciclo, diverso da quello di lettura del sensore.

SensorEvent passato al listener rimane di proprietà del SensorManager: mai tenere riferimenti all'evento che restano anche dopo il ritorno da onSensorChanged()

SensorManager potrebbe usare una pool di SensorEvent: per evitare di fare una new per ogni lettura, in questo caso gli stessi oggetti vengono riciclati più volte