Desenvolupament d'aplicacions basades en l'Android

PID_00245989

Robert Ramírez Vique
Helena Boltà Torrell

Temps mínim de dedicació recomanat: 6 hores

Universitat Oberta de Catalunya

Índex

Introducció

Aquest mòdul presenta una introducció al desenvolupament d'aplicacions mòbils basades en l'Android. Per a estudiar la plataforma Android ens centrarem en el tipus d'aplicacions natives, i així podrem veure amb més detall les peculiaritats per a aquest cas particular, i de passada tindrem una visió més clara del desenvolupament d'aplicacions natives.

L'Android va néixer com una novetat amb molt futur però ha passat de ser una gran promesa a un dels grans de la indústria. Amb molts dispositius, i creixent sense parar, és una tecnologia ben consolidada. A més, ha estat construïda amb unes bases molt sòlides, i ha intentat no tancar la porta a cap opció de desenvolupament ni arquitectura, la qual cosa és molt útil per a poder aprendre els desenvolupaments mòbils.

Farem un recorregut sobre les diferents fases de desenvolupament d'una aplicació mòbil aplicades al cas de l'Android.

Primer veurem una petita introducció a la història i les peculiaritats de la tecnologia, passant per les raons que fan avui dia de l'Android un dels dominadors mundials.

Després veurem amb més detall les eines que conformen tot l'ecosistema, veient en detall l'arquitectura de les aplicacions i el cicle de vida típic que tenen.

A continuació ens centrarem en les diferents funcionalitats que ens ofereix la tecnologia per a accedir al maquinari dels dispositius, passant per la construcció d'interfícies d'usuari, i arribant al desament de dades o les comunicacions. Aprofitant aquest apartat veurem les opcions que ens ofereix la plataforma per a poder posar a prova i millorar la nostra aplicació.

Finalment tocarem tots els temes relacionats amb les fases de distribució de l'aplicació. Veurem com es pot arribar a publicar una aplicació per a ser comprada pels usuaris, i el seguiment posterior, i també com podem aprofitar les eines que ofereix l'Android per a fer negoci no solament amb la venda directa.

Amb això tindrem una visió global de la tecnologia, suficient per a poder començar un nou projecte d'Android sabent exactament quines opcions hi ha en cada apartat o bé sabent on podem trobar la informació.

Finalment, hi ha alguns temes que no quedaran coberts en aquest mòdul, per diverses raons:

- No es veurà cap tutorial, i per tant no hi haurà la descripció de com cal fer pas per pas un desenvolupament. Per a això es proporcionaran recursos existents fora del mòdul i específicament dissenyats per a això.

- No es presentarà una guia de referència exhaustiva del desenvolupament en l'Android, ja que per a això hi ha les referències oficials, que són molt més àmplies i actualitzades. Malgrat tot, s'expliquen casos concrets per a comprendre'l millor.

- Es pressuposen bases de programació i de patrons de disseny de programari (*software*), ja que això ja ho ha assumit l'estudiant gràcies a altres assignatures.

Objectius

Amb l'estudi d'aquest mòdul es pretén que l'estudiant aconsegueixi els objectius següents:

1. Saber què és l'Android i quines són les possibilitats que té.

2. Saber com funcionen les aplicacions en l'Android, i dominar les bases per a poder desenvolupar-les coneixent les principals eines que hi ha.

3. Tenir una visió global de totes les possibilitats que ofereix la plataforma, tant en l'àmbit d'arquitectures com en l'àmbit d'API.

4. Conèixer el procés de comercialització d'una aplicació de l'Android, per a poder dur-lo a terme en cas de ser necessari, i el seguiment posterior.

1. Introducció a l'Android

1.1. Passat, present i futur de l'Android

L'Android va ser iniciat dins d'una empresa, que va ser qui li va donar el nom, Android Inc., fundada el 2003. El 2005 Google va comprar aquesta empresa, i va incorporar a les seves files alguns dels principals valedors, entre els quals hi ha Andy Rubin, Rich Miner i Chris White.

El novembre del 2007 l'Open Handset Alliance surt a la llum amb l'objectiu de desenvolupar estàndards per a dispositius mòbils. I en aquesta data descobreixen el seu nou producte, l'Android, una plataforma construïda sobre el nucli Linux.

Open Handset Alliance

L'Open Handset Alliance és un consorci de diverses empreses que inclou Broadcom Corporation, Google, HTC, Intel, LG, Marvell Technology Group, Motorola, Nvidia, Qualcomm, Samsung Electronics, Sprint Nextel, T-Mobile i Texas Instruments entre moltes altres.

El desembre del 2008 s'afegeixen a l'Open Handset Alliance catorze nous membres, incloent-hi ARM Holdings, Atheros Communications, Asustek Computer Inc., Garmin Ltd., PacketVideo, Softbank, Sony Ericsson, Toshiba Corp. i Vodafone Group Plc.

El maig del 2008 es concedeixen els premis del primer *Android developer challenge*, que és un concurs per a aconseguir les millors aplicacions per a l'Android. Amb aquest concurs, i les edicions següents, s'aconsegueix una gran repercussió i grans aplicacions.

El setembre del 2008 s'allibera la primera versió de la plataforma, la 1.0, però encara no hi havia cap dispositiu.

Finalment l'octubre del 2008 apareix el primer dispositiu, T-Mobile o HTC Dream, venut inicialment als Estats Units i posteriorment a la resta del món. I el nombre de dispositius no ha deixat de créixer, i ja són milers de dispositius diferents.

Actualment es parla de més d'un milió i mig d'activacions de dispositius diàriament amb la plataforma Android, i d'uns sis milions de desenvolupadors que ja han creat milions d'aplicacions, i no paren de créixer. S'estima que cap al 2020 hi haurà catorze milions de desenvolupadors d'aplicacions mòbils, molts dels quals treballaran en la plataforma Android.

Aquesta gran quantitat de novetats i la velocitat amb què apareixen provoca que hi hagi moltes versions disponibles a cada moment, i que hem de tenir en compte quan desenvolupem per a l'Android.

Google I/O

Les novetats del sistema operatiu Android s'anuncien en la conferència anual de desenvolupadors de Google, Google I/O.

1.2. Què és l'Android?

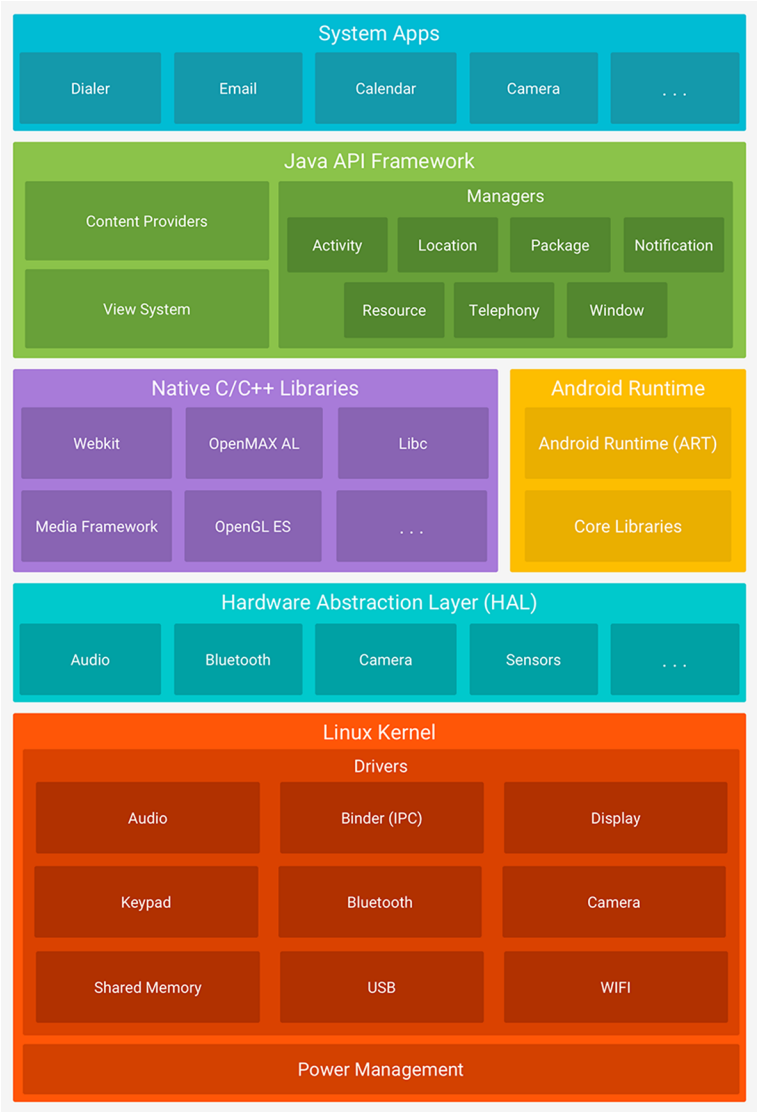
L'Android és una solució completa de programari per a dispositius mòbils. Inclou tota una pila d'aplicacions, des del sistema operatiu fins a programari intermediari (*middleware*) i aplicacions clau. També inclou eines per a desenvolupar a la plataforma, principalment utilitzant el llenguatge de programació Java, tot i que també es poden fer servir altres llenguatges com ara C++ o Kotlin. Tot això amb llicència de codi lliure Apache, la qual cosa obre moltes possibilitats.

Kotlin

Kotlin és un llenguatge de programació orientat a objectes que es pot executar sobre una màquina virtual Java. El 2017 Google va anunciar suport oficial per al desenvolupament d'aplicacions Android utilitzant Kotlin.

L'Android ens proporciona una sèrie de funcionalitats organitzades tal com es pot veure al gràfic següent:

Arquitectura de components de l'Android



1.2.1. Nucli (*kernel*)

El nucli o *kernel* de l'Android està basat en el nucli Linux, amb algunes modificacions per a adaptar-lo a les necessitats de l'Android.

Per sobre del nucli hi ha una capa d'abstracció del maquinari anomenada HAL a la que han d'accedir les aplicacions, de tal manera que si es necessita un element de maquinari (*hardware*), l'aplicació simplement demanarà el recurs en genèric, sense haver de conèixer el model exacte de maquinari. Per exemple, si volem accedir a la càmera, no importarà el tipus, la definició ni el fabricant; simplement accedirem a la càmera per mitjà del controlador corresponent.

HAL

Acrònim de capa d'abstracció del maquinari, en anglès, *Hardware Abstraction Layer*.

1.2.2. Biblioteques

La capa que se situa just sobre el HAL la componen les **biblioteques natives de l'Android**. Aquestes biblioteques estan escrites en C o C++ i compilades per a l'arquitectura de maquinari específica del telèfon, tasca que normalment fa el fabricant, que també s'encarrega d'instal·larlos al terminal abans de posar-lo en venda. La comesa que té és proporcionar funcionalitat a les aplicacions, per a tasques que es repeteixen amb freqüència, per a evitar haver de codificar-les cada vegada i garantir que es duen a terme de la manera més eficient.

Aquestes són algunes de les biblioteques que s'inclouen habitualment:

- **Open Max AL**: és un conjunt d'interfícies de programació en llenguatge C que proporciona funcionalitat per al processament d'imatges, àudio i vídeo. Està dissenyat per a usar els recursos de forma molt eficient, especialment indicat en dispositius de baixa gamma.

- ***OpenGL for embedded systems*** **(OpenGL ES)**: motor gràfic 3D basat en les API[1] d'OpenGL ES 3.1. Utilitza acceleració de maquinari (si el dispositiu en proporciona) o un motor de programari altament optimitzat quan no n'hi ha.

[(1)] interfície de programa d'aplicació o *application program interface*

- **Media Framework**: API d'Android per a la reproducció de mitjans (imatges, vídeo i àudio): MediaPlayer i MediaCodec.

- **WebKit**: motor web utilitzat pel navegador (tant com a aplicació independent com embegut en altres aplicacions). És el mateix motor que utilitzen el Google Chrome i el Safari (el navegador d'Apple, tant en OSX com en iOS).

- **Biblioteca C de sistema (libc)**: està basada en la implementació de la Berkeley Software Distribution (BSD), però optimitzada per a sistemes Linux embeguts. Proporciona funcionalitat bàsica per a l'execució de les aplicacions.

1.2.3. Entorn d'execució

L'entorn d'execució de l'Android, encara que es recolza en les biblioteques enumerades anteriorment, no es considera una capa en si mateixa, ja que també està format per biblioteques. Es basa en funcionalitats del nucli com són el *threading* o la gestió de memòria a baix nivell.

El component principal de l'entorn d'execució d'Android és la màquina virtual **ART** (Android Runtime). Inicialment es feia servir **Dalvik**, component que executava totes i cadascuna de les aplicacions no natives d'Android. Les aplicacions es compilen en un format específic per a la màquina virtual ART, que és la que les executa. Això permet compilar una única vegada les aplicacions i distribuir-les ja compilades tenint la total garantia que es podran executar a qualsevol dispositiu Android que disposi de la versió mínima del sistema operatiu que requereixi cada aplicació.

Dalvik i ART

A partir de la versió d'Android 5.0 Lollipop, ART va substituir Dalvik com a entorn d'execució.

No s'ha de confondre ART amb la màquina virtual Java, ja que són màquines virtuals diferents. De fet, Google va crear aquesta màquina virtual entre altres coses per evitar problemes de llicències. Java és únicament el llenguatge de programació, i els programes generats no són compatibles en l'àmbit de *bytecode* Java.

Bytecode

S'entén per *bytecode* el codi format per instruccions executables independents del maquinari final de la màquina.

Els arxius generats per a aquesta màquina virtual, arxius *.dex*, són molts més compactes (fins a un 50%) que els generats per a la màquina virtual Java tradicional. Per a aconseguir això ART es basa en registres en lloc d'una pila per a emmagatzemar les dades, la qual cosa requereix menys instruccions. Això permet execucions més ràpides en un entorn amb menys recursos.

Les aplicacions de l'Android s'executen cadascuna en la seva pròpia instància de la màquina virtual ART, i s'eviten així interferències entre aquestes, i tenen accés a totes les biblioteques esmentades abans i, per mitjà seu, al maquinari i a la resta de recursos gestionats pel nucli.

1.2.4. Java API Framework

La capa següent la formen totes les funcions del sistema operatiu que estan disponibles mitjançant una API escrita en Java. Òbviament, es recolzen en les biblioteques i en l'entorn d'execució que ja hem detallat. Entre les més importants hi ha les següents:

- **Administrador d'activitats (*activity manager*)**: s'encarrega de controlar el cicle de vida de les activitats i la pila d'activitats mateixa (proporcionant retrocés de navegació).

- **Administrador de finestres (*windows manager*)**: s'encarrega d'organitzar el que es mostra en pantalla, creant superfícies que poden ser «omplertes» per les activitats.

- **Proveïdor de continguts (*content provider*)**: permet que les *apps* accedeixin a dades des d'altres *apps* o comparteixin les seves pròpies dades. Per exemple, un dels proveïdors de contingut existents permet que les apli-

cacions accedeixin als contactes emmagatzemats al telèfon. Així doncs, aquesta API ens permet crear també els nostres propis proveïdors per a permetre que altres aplicacions accedeixin a informació que gestiona la nostra.

- **Vistes (*views*)**: si abans equiparàvem les activitats amb les finestres d'un sistema operatiu de PC, les vistes les podríem equiparar amb els controls que se solen incloure dins d'aquestes finestres. L'Android proporciona nombroses vistes amb què podem construir les interfícies d'usuari: botons, quadres de text, llistes, etc. També en proporciona altres de més sofisticades, com un navegador web o un visor de Google Maps.

- **Administrador de notificacions (*notification manager*)**: proporciona serveis per a notificar a l'usuari quan alguna cosa requereixi la seva atenció. Normalment les notificacions es fan mostrant una alerta a la barra d'estat, però aquesta biblioteca també permet emetre sons, activar el vibrador o fer parpellejar els LED del telèfon.

- **Administrador de paquets (*package manager*)**: les aplicacions de l'Android es distribueixen en paquets (arxius *.apk*) que contenen tant els arxius *.dex* com tots els recursos i arxius addicionals que necessiti l'aplicació, per a facilitar-ne la baixada i instal·lació. Aquesta biblioteca permet obtenir informació sobre els paquets actualment instal·lats al dispositiu Android, a més de gestionar la instal·lació de nous paquets.

- **Administrador de telefonia (*telephony manager*)**: proporciona accés a la pila de maquinari de telefonia del dispositiu Android, si en té. Permet fer trucades o enviar i rebre SMS/MMS, encara que no permet reemplaçar o eliminar l'activitat o activitat que es mostra quan una trucada està en curs (per motius de seguretat).

- **Administrador de recursos (*resource manager*)**: proporciona accés a tots els elements propis d'una aplicació que s'inclouen directament en el codi: cadenes de text traduïdes a diferents idiomes, imatges, sons i fins i tot estructuracions de les vistes dins d'una activitat (*layouts*). Permet gestionar aquests elements fora del codi de l'aplicació i proporcionar diferents versions segons l'idioma del dispositiu o la resolució de pantalla que tingui, per a poder contrarestar la fragmentació de dispositius actual.

- **Administrador d'ubicacions (*location manager*)**: permet determinar la posició geogràfica del dispositiu Android (usant el GPS o les xarxes disponibles) i treballar amb mapes.

1.2.5. Aplicacions

La capa superior d'aquesta pila programari la formen, com no podria ser de cap altra manera, les aplicacions. En aquest sac s'inclouen totes les aplicacions del dispositiu, tant les que tenen interfície d'usuari com que no, tant les natives (programades en C++) com les administrades (programades en Java[2]), tant les que vénen de sèrie amb el dispositiu com les instal·lades per l'usuari.

(2)Es disposa d'algunes de les biblioteques del JDK estàndard de Java però no de totes, i algunes estan especialment redissenyades per a ser executades en entorns mòbils.

Aquí hi ha també l'aplicació principal del sistema: Inici (*Home*), també anomenada de vegades *llançador* (*launcher*), perquè és la que permet executar altres aplicacions, proporciona la llista d'aplicacions instal·lades i mostra diferents escriptoris on es poden col·locar accessos directes a aplicacions o fins i tot petites aplicacions incrustades o *widgets*, que són també aplicacions d'aquesta capa.

El principal que cal tenir en compte d'aquesta arquitectura és que totes les aplicacions, tant si són les natives de l'Android com les que proporciona Google, les que inclou de sèrie el fabricant del telèfon o les que instal·la després l'usuari, utilitzen el mateix marc d'aplicació per a accedir als serveis que proporciona el sistema operatiu. Això implica dues coses: que podem crear aplicacions que usin els mateixos recursos que usen les aplicacions natives (res no està reservat o és inaccessible) i que podem reemplaçar qualsevol de les aplicacions del telèfon per una altra que triem.

Aquest és el verdader potencial de l'Android i el que el diferencia de la seva competència: control total per part de l'usuari del programari que s'executa al seu telèfon.

Enllaç d'interès

Per a saber més sobre què és l'Android podeu consultar aquesta adreça d'Internet:
https://developer.android.com/about/index.html

2. Fonaments de les aplicacions

Com hem vist anteriorment una aplicació Android sol ser una aplicació escrita en Java i compilada en un fitxer de tipus *.apk*. Aquest fitxer es considera una aplicació, i és l'element usat pels dispositius Android per a instal·lar l'aplicació.

Una aplicació pot estar composta per un o més components, els quals fan funcions diferents per a donar el comportament a l'aplicació, i cadascun pot ser activat de manera individual.

Veurem més endavant per quins estats pot passar la nostra aplicació, i sabrem per a què serveix cadascun i com preparar-nos-hi.

Manifest d'una aplicació

Un element important de les aplicacions de l'Android és el seu manifest, que declara la metainformació sobre l'aplicació.

És important tenir present que les aplicacions de l'Android s'executen sobre un únic i separat procés Linux que té la seva pròpia màquina virtual, de manera que tenen els recursos restringits i controlats, cosa que és ideal per a maximitzar la seguretat. I seguint el principi dels mínims privilegis, a cada procés s'adjudiquen, per defecte, els permisos d'accés als components que requereix i cap més.

Malgrat això es poden fer excepcions:

- Es poden tenir diverses aplicacions compartint un recursos; per a això han de tenir el mateix *user ID*, per la qual cosa és necessari que estiguin signades amb el mateix certificat.

- Si una aplicació sap per endavant que vol accedir a recursos com la càmera o el Bluetooth, ho ha de demanar a l'usuari. En versions antigues (Android 5.1 o anterior), aquesta consulta es fa explícitament en el moment d'instal·lar-la, i en les versions posteriors (a partir d'Android 6.0), es fa en temps d'execució en el moment d'accedir al recurs.

2.1. Components d'una aplicació

Per a facilitar la reutilització de codi i agilitar el procés de desenvolupament, les aplicacions d'Android es basen en components. Els components poden ser de quatre tipus:

Exemple d'ús de components

Si una aplicació vol utilitzar una llista amb barra de desplaçament pot utilitzar aquest component "llista" sense haver de duplicar el codi.

1) Activitats (*activity*). Com hem vist són interfícies visuals que esperen alguna acció de l'usuari. Una aplicació pot tenir una activitat o més, i des d'una activitat se'n poden invocar d'altres i tornar a l'original. Totes les activitats es-

tenen la classe activitat. El contingut visual de cada activitat el proporcionen una sèrie d'objectes derivats de la classe *View*. L'Android proporciona molts d'aquests objectes predissenyats, com botons, selectors, menús, etc.

2) Serveis (*services*). Els serveis no tenen interfície gràfica; són processos que s'executen en segon pla per a fer accions de llarga durada.

Exemple de serveis

Un exemple és la reproducció d'una cançó. Per a una aplicació reproductora podríem tenir diverses activitats per a mostrar llistes de cançons o un reproductor amb botons, però l'usuari esperarà que la cançó continuï sonant encara en sortir de l'aplicació (acabar l'activitat), per la qual cosa aquesta aplicació haurà de controlar un servei perquè es reprodueixi la música. Qualsevol servei estén la classe *Service*.

3) Proveïdors de continguts (*content providers*). Permet que una aplicació posi certes dades a disposició d'altres aplicacions; aquestes dades poden estar desades en el fitxers, en una base de dades SQLite, en el Web o qualsevol altre lloc que sigui accessible. Per mitjà d'això altres aplicacions poden modificar aquestes dades o preguntar-hi.

Per exemple, un enregistrador de sons pot compartir aquestes dades amb un reproductor de música. Aquestes dades es poden emmagatzemar en el sistema de fitxers o en bases de dades. Per a proveir continguts, s'ha d'estendre la classe *ContentProvider*.

4) Receptors d'esdeveniments (*broadcast receivers*). Aquests components simplement estan escoltant que es produeixin determinats esdeveniments (bateria baixa, canvi de l'idioma del dispositiu, baixada d'una imatge nova, etc.). Qualsevol aplicació pot tenir tants receptors per a tants esdeveniments com vulgui, i cadascun ha d'estendre la classe *BroadCastReceiver*.

Un exemple és una galeria d'imatges que indexa en l'aplicació qualsevol imatge que l'usuari baixi mitjançant el navegador o una altra aplicació, per la qual cosa està escoltant l'esdeveniment de baixada d'imatge.

Com es pot veure hi ha diverses maneres d'accedir a una aplicació, i comunicar-se entre les aplicacions. Aquest punt és sens dubte un dels punts forts de les aplicacions de l'Android. Per a poder comunicar-se entre aplicacions, i com que s'executen en processos amb permisos diferents, no es pot fer directament, i per a això s'utilitzen les intencions o *intents*.

2.2. Cicle de vida

El cicle de vida d'una aplicació està íntimament lligat al d'una activitat, i el seu cicle de vida també. Per a conèixer millor com funcionen les aplicacions veurem com s'organitzen en l'àmbit d'usuari.

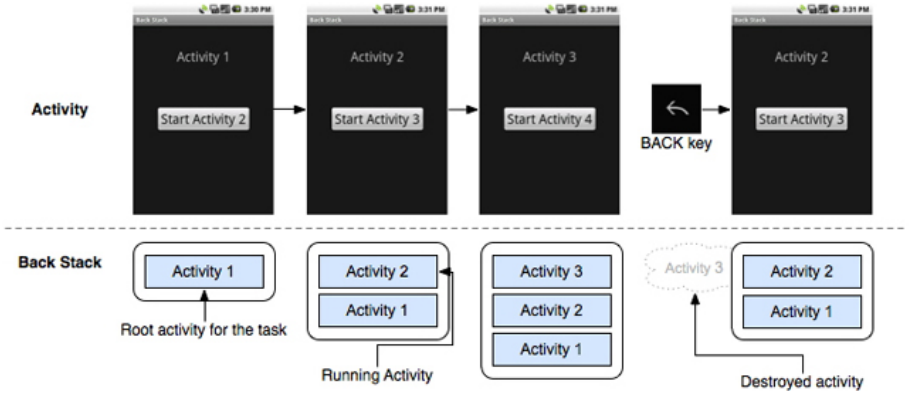
Un concepte important en l'Android són les **tasques** o ***tasks***. Una tasca conté una o diverses activitats, i d'alguna manera és la tasca d'usuari per a aconseguir alguna cosa.

Exemple de tasca

Per a poder escriure un correu electrònic tindrem:

- L'activitat de gestor de correu electrònic, per a veure la llista.
- L'activitat del client de correu electrònic; després se seleccionarà compondre un correu nou.
- Posteriorment, per a triar el destinatari, l'activitat del selector de contactes.
- I si volem afegir una fotografia, l'activitat de la galeria.

Piles de tasques i d'activitats i les seves transicions



Font: http://developer.android.com/

Es pot sortir d'una tasca prement el botó Enrere (*Back*) o bé amb el botó Inici (*Home*). En el cas del botó *enrere* es destruirà la tasca i totes les activitats, i per tant si torna a aquesta activitat s'haurà perdut l'estat (llevat que es desi explícitament). En canvi, si se surt de la tasca per mitjà del botó Inici (*Home*) l'estat de l'activitat està implícitament desat.

Per tant, una tasca té diverses activitats, i la raó és que una activitat pot invocar una altra activitat, i totes són en la mateixa tasca. Quan una activitat en crea una altra ho fa per mitjà d'un *intent*, i ho pot fer de dues maneres: esperant resultats (*startActivityForResult()*) o no esperant-los (*startActivity()*). En tots dos casos, una vegada finalitzada amb l'activitat creada, es torna a l'activitat creadora. Això es pot produir moltes vegades gràcies a les piles d'activitats dins d'una tasca. Aquestes tasques també tenen la seva pila per a permetre interrupcions i en certa manera permetre múltiples fils d'execució.

Múltiples fils d'execució

Múltiples fils d'execució o *multithreading*, aplicat a la indústria mòbil, vol dir poder mantenir diverses aplicacions executant-se sense tancar-ne cap de les dues.

Exemple

Si estem treballant amb una aplicació de calendari i veiem en una cita que hi ha una adreça electrònica i volem enviar un correu electrònic a aquest compte, però en aquest punt ens truquen al telèfon.

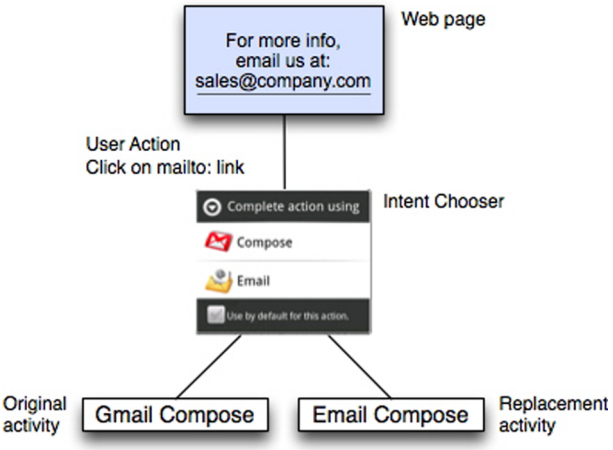
Com a resultat, a la pila de tasques tindrem dues tasques, una l'aplicació de calendari, i una altra la corresponent a l'aplicació de telèfon, a causa que la segona ha interromput la primera.

A més la primera tasca tindrà diverses activitats, en concret la llista d'elements del calendari, l'activitat per a veure un element del calendari concret, i finalment l'activitat que correspon a una altra aplicació, que és la de tramesa de correu electrònic.

Quan finalitzi la tasca de telèfon tornarem a la tasca anterior i amb l'activitat de correu electrònic.

Per tot això les activitats s'han de comunicar entre si, i invocar una activitat sembla una cosa bastant habitual, i això és un gran punt a favor de l'Android. Però d'altra banda es defensa fer que les activitats estiguin tan poc acoblades entre si com sigui possible. Això s'aconsegueix amb els *intents*.

Hi ha activitats que poden tenir diversos punts d'entrada i per a això defineixen diversos *IntentFilters*. Amb aquests *IntentFilters* també es poden fer activitats que esperin intencions conegudes, com pot ser enviar un correu electrònic o triar una fotografia. Si hi ha més d'una activitat per a un mateix *intent*, ens preguntarà quina activitat cal iniciar en produir-se aquesta intenció. La informació dels *intents* als quals reaccionarà una activitat està definida en el manifest.



Com veiem, una activitat pot estar en diversos estats mentre s'està executant, que es poden resumir bàsicament en tres estats diferents:

1) Resumida. L'activitat és en primer pla i té el focus de l'usuari. També es coneix com a *estat d'execució* o *running*.

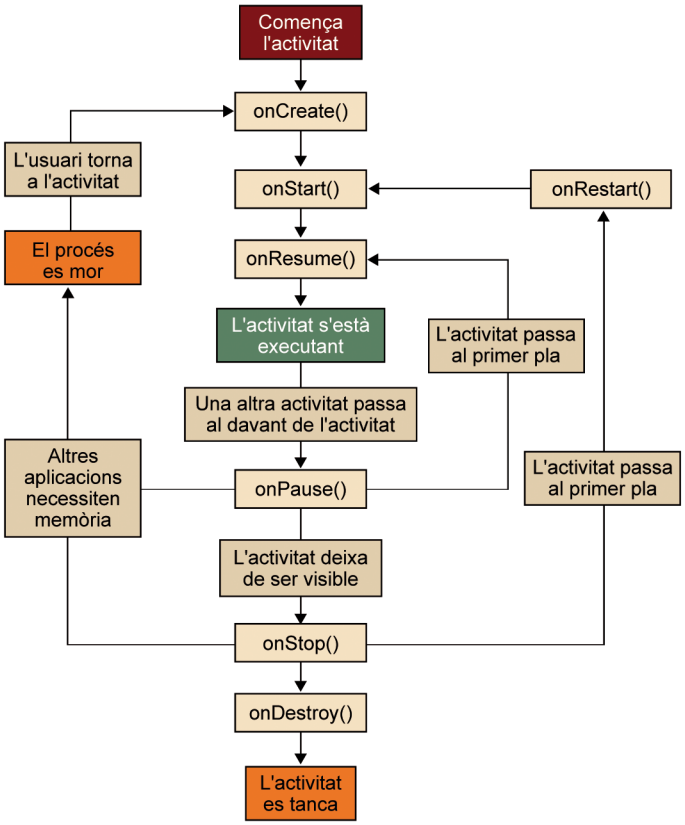
2) Pausada. Hi ha una altra activitat en primer pla i està *resumida*, però la pausada és encara visible. Això pot succeir quan l'activitat en primer pla no cobreix tota la pantalla o bé és parcialment transparent. L'activitat manté tota la informació de memòria i és completament viva, encara que pot arribar a ser destruïda en cas extrem de falta de memòria.

3) Acabada. L'activitat està totalment ocultada per una altra activitat, i està en segon pla. L'activitat és totalment viva, però no és visible i pot ser destruïda quan el sistema necessiti més memòria.

Les activitats pausades o acabades el sistema les pot destruir avisant (cridant al mètode *finish()*) o directament destruint el procés de sistema. Quan una activitat torna després de ser destruïda, es construeix novament.

Per a aconseguir programar les transicions d'estats de manera flexible i mantenible, l'Android proveeix un sistema de repetició de trucada o *call back*, de manera que el desenvolupador pot prendre les accions apropiades en cada moment. A continuació podem veure a la figura següent quines són:

Cicle de vida d'una activitat en l'Android



Font: http://developer.android.com/guide

Hi ha alguns esdeveniments que són especialment importants:

1) onCreate. És el punt d'inici, on hem d'inicialitzar tota la interfície gràfica a més de crear tots els elements. Com que s'hi pot arribar després d'haver estat destruïda, és possible que hàgim de recuperar l'estat d'algun origen persistent.

2) onPause i **onStop**. Són els esdeveniments corresponents a les parades de l'aplicació, i s'ha de desar l'estat de la sessió corresponent. Després d'aquests esdeveniments l'aplicació passa a estar pausada (*onPause*) o acabada (*onStop*).

3) onResume i **onStart**. L'aplicació torna de l'estat pausada o acabada i ha de recuperar la informació de la sessió.

4) onDestroy. Són el punt on s'han d'alliberar els recursos, i és just abans de destruir definitivament l'aplicació.

Quant al cicle de vida és important tenir en compte que una vegada s'ha passat l'estat *onPause* (sempre succeeix abans d'un *onStop* i un *onDestroy*) l'aplicació pot ser destruïda directament pel sistema operatiu. Per tant, la manera segura d'emmagatzemar la informació de l'estat de la sessió és en el mètode *onPause*, però cal fer-ho amb compte, perquè estem bloquejant l'activitat següent i podem empitjorar l'experiència d'usuari.

Per a facilitar la vida al desenvolupador, hi ha un mètode al qual crida el sistema quan creu que és necessari desar l'estat: *onSaveInstanceState()*. Totes les vistes l'implementen amb una implementació per defecte; per exemple, un camp de text en desa el valor. Es basa a desar parells de clau i valor per a poder ser recuperats *a posteriori*. En el cas de les activitats succeeix el mateix, i es pot sobreescriure el comportament per defecte per a desar l'estat, i l'objecte on es desen les claus-valor passarà al mètode *onCreate* quan es recuperi l'estat després de ser destruït; pot utilitzar també el mètode *onRestoreInstanceState*.

Tot aquests mètodes utilitzats per a gestionar la destrucció i la creació de les activitats tenen especial interès a causa que quan hi ha canvis en la configuració (com l'orientació de la pantalla, l'idioma, l'aparició del teclat virtual) el sistema reinicia l'activitat (es crida a *onDestroy* i després *onCreate*). Això és per a facilitar l'aplicació a adaptar-se a les noves configuracions.

2.3. Intencions (*intents*)

Per a poder-se comunicar entre diverses aplicacions d'una mateixa tasca, es llança una **intenció** (*intent*) o sol·licitud perquè un component dugui a terme una tasca.

Els *intents* ofereixen un servei de pas de missatges que permet interconnectar components d'aquesta o de diferents aplicacions.

Els *intents* s'utilitzen per a invocar una nova activitat o bé per a enviar esdeveniments a múltiples activitats (anomenats *broadcast intents*). Els *intents* són gestionats pels receptors d'esdeveniments o *broadcast receivers*, que pot escoltar qualsevol *intent*.

Exemple d'*intent*

L'Android llança un *intent* amb els canvis de bateria, canvis de connectivitat o missatges entrants per a tots els receptors d'aquests *intents*.

Un *intent* es descriu amb els atributs següents:

- El nom del component al qual volem avisar. Per exemple, l'aplicació de correu electrònic.
- L'acció que es vol llançar. Per exemple, editar, trucar, sincronitzar o informació de bateria baixa.
- Les dades sobre l'acció. Per exemple, escriure un nou correu.
- La informació extra. L'adreça de correu del destinatari i el títol.

Aquests *intents* es poden invocar de dues maneres:

- **Explícita**. S'especifica explícitament en codi quin component és l'encarregat de gestionar l'*intent*.

Exemple

```
Intent intent = new Intent(Context, Activity.class);
startActivity(intent);
```

- **Implícita**. És la plataforma la que determina, per mitjà del procés de resolució d'*intents*, quin component és el més apropiat per a manejar l'*intent*.

Exemple

```
Intent intent = new Intent(Intent.ACTION_DIAL,
    URI.parse(tel:928-76-34-26));
startActivity(intent);
```

Un component declara la seva capacitat d'atendre un *intent* per mitjà del manifest, afegint etiquetes <*intent-filter*>.

2.4. Proveïdors de continguts (*content providers*)

La manera d'accedir o de compartir informació amb altres aplicació és per mitjà dels proveïdors de continguts (*content providers*). No hi ha cap zona comuna, ja que cada aplicació té el seu espai d'execució i permisos propis.

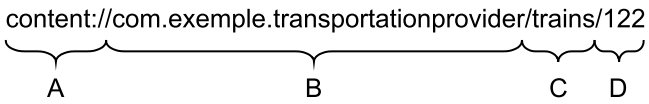
Aquests proveïdors ens poden permetre, amb les dades, accedir-hi, modificar-les o fins i tot esborrar-les, sempre depenent del sistema de permisos que implementa l'Android. A més podrem oferir les nostres dades a altres aplicacions o bé incloure aquesta informació en algun proveïdor ja existent o implementant-ne un de propi i donar-li visibilitat.

Realment no treballarem mai amb un proveïdor concret sinó que utilitzarem classes intermèdies, com per exemple la classe *ContentResolver*. Per a aconseguir la instància que ens interessa simplement cridarem al mètode de la nostra activitat *getContentResolver()*, amb la qual cosa tindrem accés per a poder fer les accions d'inserir, modificar, esborrar i consultar la informació proporcionada pels proveïdors. Per al cas de consulta de dades també hi ha l'opció d'utilitzar el mètode *managedQuery*, que és equivalent al mètode del *ContentResolver.query*.

URI

Un identificador uniforme de recursos o *universal resource identifier* (URI) és un identificador únic dins de la nostra aplicació. Els URL són un tipus d'URI.

Les dades que intercanviarem seran en forma d'una única taula, independentment de quina en sigui la representació interna. Cada columna tindrà un nom i un tipus coneguts, per a poder tractar amb les dades (el nom de la columna). Cada fila de la taula tindrà un identificador únic, el nom de columna del qual serà *_ID*. A més hi ha uns URI que serviran per a identificar cadascun dels recursos o conjunt de recursos del proveïdor de continguts. Aquests URI tindran la forma següent:



en què:

- *A* és la part fixa per a identificar que es tracta d'un proveïdor.
- *B* és el nom de la classe *Provider*. És necessari declarar-la en el manifest.
- *C* és la ruta del proveïdor de continguts, que pot ser nul·la si només es proveeix informació d'un tipus, o bé complexa, com es vulgui, si hi ha diversos tipus, com per exemple *trains/bcn* o *trains/mdm*.
- *D* identifica una fila de les dades de manera única.

Per a poder fer una consulta de dades, tant el mètode *managedQuery()* com el *query()* reben els paràmetres següents:

- **L'URI que hem vist abans**. Si l'última part (*D*) és present es consultarà únicament una fila.
- **Nom de les columnes que volem aconseguir**. Si aquest paràmetre és nul significa que volem totes les dades.
- **Un filtre a l'estil SQL**. Seria tot el que pot venir en la sentència *WHERE*, però sense la paraula *WHERE* en concret. Si és nul es lliuraran totes les files.
- **Una sèrie d'arguments de selecció**. Igual com en altres llenguatges de consulta SQL, dins dels nostres filtres podem afegir caràcters interrogants (?) que només seran coneguts en temps d'execució. Aquí posarem els valors corresponents a aquests interrogants en cas d'existir.
- **Un criteri d'ordenació**. Igual que abans, de la mateixa manera que en l'SQL, es pot definir ordre ascendent o descendent.

Per a modificar les dades sempre haurem d'usar els mètodes del *ContentResolver*. Així hi podrem passar un URI, general si és inserció o modificació massiva, i concret d'una fila si es tracta d'actualitzacions o esborrament. A més, treballarem amb objectes *ContentValues* per a passar les dades que volem modificar.

Enllaç d'interès

Per a saber més sobre els proveïdors de continguts podeu visitar aquesta pàgina d'Internet:
http://developer.android.com/guide/topics/providers/content-providers.html

2.5. Manifest

El manifest de l'Android és un fitxer XML que defineix informació sobre la nostra aplicació. Aquest fitxer es troba en l'arrel del nostre projecte Android, i és imprescindible perquè la nostra aplicació sigui executada pel sistema operatiu.

Serveix per a declarar una sèrie d'informació sobre la nostra aplicació que el sistema ha de conèixer per endavant:

- Permisos necessaris per a poder funcionar, com l'accés a la càmera.
- El nivell de l'API de l'Android necessari per a treballar.

  Nivell de l'API de l'Android

  Els nivells de les API de l'Android es refereixen a la versió que ha estat distribuïda. Aquests nivells difereixen de les versions de la plataforma conegudes comercialment i són un nombre enter consecutiu; així, la versió 1.0 té el nivell 1, però la 7.1 té el 25. Serveixen per a identificar quines API o quines versions hi ha disponibles.

- Les característiques de maquinari i programari necessàries per a treballar, com per exemple tenir o no càmera.
- Les API a què necessita accedir, a part de les que estan incorporades en l'Android *framework* API. Per exemple, la biblioteca del Google Maps.
- Els components que formen la nostra aplicació.

2.5.1. Definició de components

Una part important i imprescindible en cadascun dels manifestos és la definició dels components que formen l'aplicació (són dins de l'element XML <*application*>). Aquests components poden ser:

- <*Activitat*>, que correspon a una activitat.
- <*Service*>, per a definir un servei en segon pla.
- <*Receiver*>, per a definir un receptor d'esdeveniments.
- <*Provider*>, per a definir un proveïdor de continguts.

En qualsevol dels casos anteriors s'ha de definir el nom (*android:name*) que defineix el nom de la classe totalment qualificada.

Aquesta gairebé és l'única manera de definir un component de la nostra aplicació i, per tant, de fer que sigui utilitzable en el cicle de vida. Hi ha un cas en el qual no és necessari fer-ho així, que és en la creació dinàmica de receptors d'esdeveniments; llavors simplement es creen objectes *BroadcastReceiver* i es registren amb el mètode *registerReceiver()*.

Com hem vist abans amb els *intents*, és possible invocar components explícitament (excepte els proveïdors de continguts), utilitzant una instància creada de la classe, o bé implícitament per mitjà del manifest. Això es fa per mitjà dels elements <*intent-filter*> del manifest, on es defineix:

- **L'acció o accions a què està associat el component**. Aquest camp és obligatori i defineix l'estructura de la resta del filtre. Aquí es defineix l'acció que volem, com pot ser *ACTION_VIEW*, *ACTION_EDIT*, *ACTION_MAIN*.
- **La categoria**. Serveix per a definir informació addicional sobre l'acció per executar.
- **El tipus**. Defineix el tipus MIME de les dades de l'*intent*.
- I alguns elements més no obligatoris.

Exemple de definició de categoria

Amb la categoria *CATEGORY_LAUNCHER* diem que el component ha d'aparèixer en el llançador o *launcher* com una aplicació.

En el cas de les activitats, un filtre molt normal és el de llançar l'aplicació; així, per exemple, podem tenir per a definir que la nostra aplicació es pot executar per mitjà del llançador el codi següent:

```
<intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

2.5.2. Definició de requisits

Una funció important que fa el manifest d'Android és definir els requisits mínims de maquinari i programari. Això serveix per a filtrar les aplicacions que es mostren a les botigues d'aplicacions, de manera que només es mostren aquelles que hi ha disponibles per al dispositiu actual. En cas d'intentar instal·lar igualment les aplicacions en un dispositiu que no compleix algun d'aquests requisits, no ens deixarà.

Aquests requisits es defineixen per mitjà d'alguns elements del manifest de l'Android, la majoria de l'estil <*uses*-*>. Alguns dels més importants són:

- **Tipus de pantalles**, que es defineixen en l'element <*supports-screens*>, i especifiquen coses com la mida de la pantalla (amb opcions com *small*, *normal*, *large* i *extra large*) o la densitat de píxels (amb les opcions *low density*, *medium density*, *high density* i *extra high density*).

- **Configuracions d'elements d'entrada**, definides en l'element <*uses-configuration*>. Si la nostra aplicació requereix algun element especial com els teclats, el ratolí de bola o l'existència d'elements de navegació direccional, ho ha d'especificar en aquesta secció.

- **Característiques del dispositiu**, definit en <*uses-features*>, que determina coses com si es necessita algun tipus de comunicació específica o algun sensor, o si requereix càmera.

- **Nivell de l'API**, definit en l'element <*uses-sdk*>. Pot definir un mínim i un màxim de les versions de l'SDK suportades.

- **Permisos**, definits en <*permission*>. Serveixen per a limitar l'accés a les parts sensibles del sistema, com *android.permission.READ_OWNER_DATA*. Aquests permisos s'han de definir perquè l'aplicació pugui accedir a aquestes parts sensibles i també es poden definir, en el manifest mateix, nous permisos.

- **Biblioteques externes**, definides en <*uses-library*>. Tindrem un element per cada biblioteca externa.

Permisos

A partir d'Android 6.0, els permisos es consulten a l'usuari en temps d'execució (quan s'hi vol accedir) i no en temps d'instal·lació.

No s'ha d'assumir en les nostres aplicacions res més que les API estàndard de l'Android; qualsevol altra cosa haurà de ser esmentada en el manifest per a assegurar-ne el funcionament.

Enllaç d'interès

Per a saber més sobre el manifest podeu visitar aquesta adreça d'Internet:

http://developer.android.com/guide/topics/manifest/manifest-intro.html

2.6. Recursos

Els recursos en l'Android són aquells fitxers que pertanyen al projecte, i poden ser de tipus multimèdia o estàtics. Segons l'organització en directoris s'utilitzen per a un tipus de dispositiu o un altre, i també segons la informació de context de l'aplicació. Dins d'aquests fitxers hi ha molts fitxers XML que serveixen per a definir *layouts*, menús, cadenes de text, estils, etc.

Aquests recursos són una part important de la plataforma perquè s'usen en diverses parts de la plataforma per a objectius diferents, però tenen el mateix funcionament. Per tant, coneixent com funcionen per a un cas, per a la resta serà idèntic.

L'*objectiu* principal d'aquests recursos és evitar-ne la gestió al desenvolupador i delegar l'elecció de quin fitxer cal utilitzar a la plataforma Android. Aquesta és una manera d'atacar la fragmentació en l'Android.

Exemple

Un bon exemple és el cas de tenir una imatge per a l'aplicació, en què la imatge ha de ser diferent per a l'aplicació amb el dispositiu en vertical i per al dispositiu en horitzontal. En aquest cas nosaltres només hem de proporcionar les dues imatges al lloc correcte i l'Android s'encarrega de gestionar quina aplicació ha d'utilitzar.

Aquests *resources*, perquè puguin ser interpretats per l'Android, han d'estar organitzats d'una manera especial, sempre dins del directori *res* de l'arrel del nostre projecte. A partir d'aquí tenim diversos subdirectoris.

Vegem-ne alguns:

- *drawable*: bàsicament fitxers d'imatges o animacions en XML.
- *layout*: fitxers que defineixen el disseny o distribució dels elements a la pantalla.
- *values*: valors de cadenes de caràcters, enters, colors, etc., que per exemple són candidats a ser internacionalitzats.
- D'altres poden ser *anim* per a fitxers XML d'animacions, *raw* per a fitxers arbitraris, o *menu* per a fitxers XML que defineixen un menú.

Aquests recursos, per a poder ser utilitzats des del codi Java, a part dels usos implícits com alguns *layouts*, la plataforma els transforma automàticament en un fitxer anomenat *R.java*. Aquesta classe Java es regenera en cada canvi dels fitxers de recursos, de manera que totes les propietats són identificades unívocament per atributs de la classe *R* de Java (per exemple, *R.layout.main*).

2.6.1. Recursos alternatius

Com hem vist, amb els recursos es pretén aïllar el desenvolupador de l'elecció del recurs adequat. Amb aquests recursos definits fora del codi es pretén aconseguir diferenciar entre textos en diferents idiomes, o diferents densitats de les pantalles.

Els recursos alternatius al recurs per defecte es defineixen amb noms de directoris alternatius, que es formen amb la formula

<resources_name>(_<config_qualifier>)+

en què *resources_name* correspon al tipus de recurs, que pot ser qualsevol dels vistos anteriorment, i *config_qualifier* serveix per a determinar una configuració diferent. Es pot definir més d'un *config_qualifier*, tots separats pel caràcter guió baix (_).

Exemple

Si tenim un fitxer *icon.png*, podem tenir el recurs per defecte a *res/drawable/icon.png*, mentre que la versió anglesa la tindrem a *res/drawable_**en**/icon.png*.

L'Android suporta diversos qualificadors de configuració. A continuació mostrem algunes de les opcions més interessants:

Evolució dels qualificadors

Hi ha algunes d'aquestes configuracions que es van ampliant al llarg de la història de l'Android per a suportar els nous tipus de dispositius, com el cas de les tauletes tàctils.

- **Operador i nació**. Serveix per a identificar l'operador que ens dóna la cobertura. Es pot identificar només al codi de nació o nació i operador. Per exemple, *mcc310* o *mcc310-mnc004*.

- **Idioma i regió**. Els idiomes amb dos possibles components separats per un guió. Primer l'idioma (segons ISO639-1) i després es defineix la regió (precedit de *r* i després el codi ISO3166-1-alpha-2). Per exemple, fr-rFR, fr-rCA.

- **Mida de la pantalla**: es defineixen quatre tipus: *small*, *normal*, *large* i *xlarge*.

- **Orientació de la pantalla**: defineix l'orientació actual, i així tenim dues opcions:
  - *port* per a orientació de retrat o vertical, i
  - *land* per a orientació de paisatge o horitzontal.

- **Mode nit**: per a indicar si som en hora de dia o de nit, útil, per exemple, per a colors. Tenim dues opcions, *night* i *notnight*.

- **Densitat de la pantalla (dpi)**: Defineix diverses opcions segons la densitat de la pantalla en nombre de píxels possibles. Per exemple:
  - *ldp (low)*: 120 dpi.
  - *mdpi (medium)*: 160 dpi.
  - *hpdi (high)*: 240 dpi.
  - *xhdpi (extra-high)*: 320 dpi.
  - *xxhdpi (extra-extra-high)*: 480 dpi.
  - *xxxhdpi (extra-extra-extra-high)*: 640 dpi.

- **Mode d'entrada de text primari**: indica el tipus de teclat de què es disposa; així podem tenir:
  - *nokeys*: no es disposa de teclat físic.
  - *qwery*: es disposa d'un teclat complet.
  - *12key*: el dispositiu disposa d'un teclat de dotze tecles.

L'ordre dels qualificadors és important, segons estan definits en la llista anterior, i amb més detall en la referència oficial de l'Android.

Els noms no són sensibles a majúscules i minúscules, i no es poden repetir els diversos valors per a un mateix qualificat.

Exemples

Amb les combinacions es poden aconseguir coses com *drawable_port_hpdi*, o *drawable_en_rUS_land*, *drawable_mcc310_en_large_port_night_nokeys*.

Cada vegada que es demana un nou recurs, en temps d'execució i per mitjà per exemple de la classe *R*, l'Android fa la cerca amb els valors de les configuracions actuals i les opcions disponibles en el directori *res*.

Perquè l'Android pugui escollir el recurs adequat utilitza un algoritme d'eliminació d'opcions disponibles, agafant cadascuna de les configuracions actuals i eliminant les que contradiuen la configuració fins que només queda una opció.

Per a poder evitar problemes de recursos no trobats, sempre és una bona pràctica donar un valor per defecte per al recurs a la carpeta i fitxer per defecte, que és on mirarà l'Android en cas de no trobar el recurs.

3. Interfície gràfica

Una de les parts importants de qualsevol aplicació sempre és la interfície gràfica amb què l'usuari ha de treballar, també coneguda com a *interfície d'usuari*. L'Android, a més, coneixent la situació de fragmentació de la seva interfície, opta per una sèrie de solucions per a aconseguir aplicacions tan fàcils de mantenir i adaptar com sigui possible.

Per a això l'Android proporciona diverses maneres de definir la nostra interfície:

- Via el fitxer *layout.xml*.
- Per mitjà de la programació dels components visuals, com es fa en la majoria d'aplicacions de taula.
- Altres tècniques, que solen ser la combinacions de les anteriors, com per exemple l'ús de *LayoutInflater*.
- Utilitzant el LayoutEditor, l'editor visual de l'IDE Android Studio.

Independentment de la manera com es construeixi la interfície gràfica, aquesta es compon dels mateixos elements, que són bàsicament o *View* o *ViewGroup*, o qualsevol classe que hereti d'aquestes.

Com moltes altres interfícies d'usuari, l'Android té un model basat en el patró de disseny observador, i per tant treballa amb esdeveniments i escoltadors o *listeners*, cosa que fa molt més senzilla la gestió i codificació.

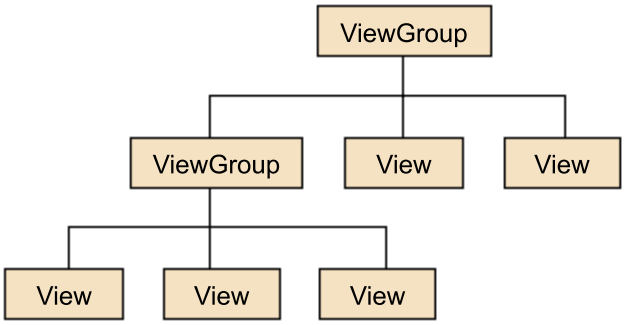
Dins de l'ecosistema de desenvolupament, sempre acostumen a aparèixer facilitats per a desenvolupar treballs laboriosos, i aquest és un d'aquests casos. Veurem també com hi ha eines per a fer el treball de dissenyar la interfície gràfica de manera més senzilla i menys propensa a errors.

3.1. Elements de la interfície gràfica

Podem distingir dos elements de la interfície gràfica, com són els *View* i els *ViewGroup*. Els primers corresponen a tots els elements gràfics, com botons, caixes de text, elements de selecció o elements per a dibuixar. Els segons són els que serveixen per a agrupar els primers. Per tant, un *ViewGroup* pot contenir altres *ViewGroups* o més *Views*.

Aquesta manera de definir les interfícies fa molt fàcil la reutilització, i tenir components autònoms.

Exemple de possible herència de *Views* i *ViewGroups*



3.1.1. *ViewGroup*

Els *ViewGroup* són contenidors per als quals l'Android proporciona una sèrie de subclasses que podem usar per a construir la nostra interfície. Entre aquestes destaquem:

- *FrameLayout*. És el tipus més simple, amb un espai en blanc en el centre que es pot omplir amb un objecte.

- *LinearLayout*. Disposa tots els fills en una sola direcció, horitzontal o vertical, un darrere de l'altre.

- *ListView*. Mostra tots els fills en una columna vertical amb barra de desplaçament.

- *TableLayout*. Disposa els fills de manera tabular, i els pot organitzar per columnes i files.

- *TabLayout*. Serveix per a dividir la nostra interfície usant pestanyes, i així podem tenir en una mateixa activitat vistes diferents organitzades per pestanyes.

- *RelativeLayout*. Disposa els elements de manera relativa a altres elements o al contenidor. És un dels contenidors més utilitzats per la versatilitat que té a l'hora d'afegir nous elements i adaptar-se a canvis de la interfície, per a coses com noves dades.

- *AbsoluteLayout*. Es defineixen els elements a partir de les seves coordenades x i y corresponents a la cantonada superior esquerra.

És recomanable intentar usar contenidors relatius sobre absoluts per a adaptar-nos millor a tot tipus de pantalles, i canvis.

En cas que cap de les subclasses anteriors no ens interessi, sempre podem fer la nostra pròpia subclasse de *ViewGroup* o estendre'n alguna ja existent.

3.1.2. *View*

Els *View* són els elements que finalment seran objectes visuals. Els objectes normals són coneguts com a *Widgets*, i aquí podem veure els elements d'un formulari com poden ser el botó, el camp de text, o elements més sofisticats com un selector de data o d'hora, o un element per a votar (*RatingBar*).

També hi ha altres elements més potents, com poden ser:

- **WebView**. És un element visual que inclou el resultat de visualitzar una pàgina web. Es pot habilitar o deshabilitar l'ús de JavaScript per a aquest element, segons ens convingui.

- **Gallery**. Serveix per a mostrar elements en una galeria horitzontal. Tenint l'element central com a seleccionat, el següent i anterior se solen poder veure. És en realitat un contenidor d'elements, però sol ser usat dins d'altres contenidors.

- **Autocomplete**. És una caixa de text, on a l'usuari li fem suggeriments d'autocompletament mentre escriu.

- **Map View**. Serveix per a visualitzar elements objectes utilitzant com a contenidor un mapa del Google Maps.

Igual que passa amb els contenidors, si no és suficient amb aquests elements podem fer les nostres pròpies adaptacions o objectes visuals totalment personalitzats.

3.1.3. *Fragments*

Els fragments (*fragments*, en anglès) representen un comportament de la interfície gràfica o només una porció d'aquest, inclòs en una activitat.

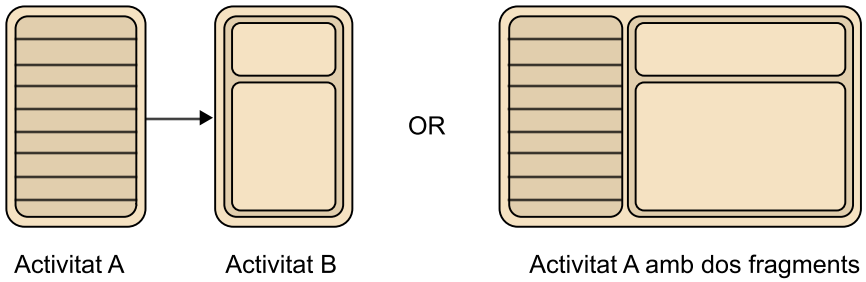
Els fragments apareixen en l'Android a partir de la versió 3.0, principalment per a suportar dissenys flexibles i dinàmics en pantalles grans, com per exemple les tauletes. I s'usen per a evitar de manera més senzilla la fragmentació.

Sempre han d'estar associats a una activitat, però els fragments poden estar definits en un fitxer XML diferent, i tenir comportament específic, de manera que reutilitzar-los pot ser molt senzill. Una cosa molt normal és incloure diversos fragments en la mateixa activitat en pantalles grans, i en canvi dividir-ho en dues activitats per a pantalles més petites.

Exemple de fragments

A la figura següent es pot veure un exemple de dos fragments dividits diferents segons la mida de pantalla.

Font: http://developer.android.com/



El cicle de vida d'un fragment està íntimament lligat al de l'activitat on està inclòs, però es poden definir accions diferents per al fragment en els esdeveniments del cicle de vida, de manera que el funcionament que té és més autònom.

Una funcionalitat interessant dels fragments és fer accions als fragments des de la nostra activitat. Aquestes accions poden ser desades a la pila d'activitats de manera que es pugui navegar cap enrere en les accions fetes. Aquestes accions es desen com a part de la classe *FragmentTransaction*. A aquesta classe s'hi han de fer les crides corresponents als mètodes *add*, *remove* o *replace* per a aconseguir els canvis que volem; per a aplicar la transacció s'ha de cridar al mètode *commit*, però si volem que aquesta acció pugui ser rebutjada per mitjà de la pila d'activitats, cal cridar al mètode *addToBackStack*.

3.1.4. Altres elements de la interfície gràfica

A més dels elements genèrics de la interfície, hi ha alguns elements o mètodes de programació que es van afegint a l'Android per a facilitar el desenvolupament.

Menús

Els menús serveixen per a permetre a l'usuari fer accions especials depenent d'on estiguin associats, si a una activitat o a una *View* concreta. Poden ser de tres tipus:

- **Menú d'opcions**. El menú que apareix quan es prem el botó menú des d'una activitat.
- **Menú contextual**. Apareix després de polsar de manera prolongada una *View*.
- **Submenú**. És el menú que apareixerà en cas necessari, quan un element d'un altre menú sigui polsat.

Aquests menús, com la resta d'element gràfics, poden ser definits usant els recursos XML o bé de manera programàtica. A més, tenen esdeveniments per a gestionar les seves accions.

Notificacions i diàlegs

També és important tractar el tema de les notificacions i diàlegs que es poden mostrar a l'usuari. Dins dels diàlegs podem generar-los per mitjà de la classe *Toast*, que són simples missatges emergents i es limiten a textos informatius, o bé per mitjà de la classe *Dialog* i les subclasses que té.

En el cas dels missatges informatius via *Toast*, bàsicament es generen cridant a *makeText* informant del context visual, per a poder mostrar el missatge on correspongui, del text i de la durada d'aquest.

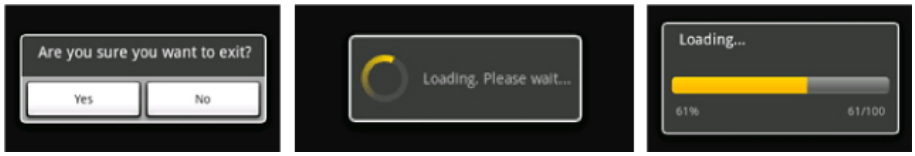
Dels diàlegs veurem algunes subclasses destacades:

- **AlertDialog**. Serveix per a mostrar informació i donem d'una a tres opcions com a acció a l'usuari o una llista d'elements per a triar, seleccionables amb caselles de selecció (*checkboxes*) o botons de selecció (*radio buttons*).

- **ProgressBar**. Mostra un diàleg que informa que l'aplicació està treballant, per a determinar-ne l'evolució. Substitueix el diàleg ProgressDialog utilitzat en versions anteriors.

- **DatePickerDialog/TimePickerDialog**. Serveix per a escollir una data.

Alguns exemples de diàlegs modals



Aquests diàlegs prenen el focus de l'aplicació i fins que no finalitzi l'acció o s'ignori el diàleg no es permet continuar.

Les notificacions, en canvi, són informacions que no han de ser ateses immediatament. Són informacions típicament generades per serveis, que informen d'esdeveniments, com per exemple l'arribada d'un correu electrònic. Aquestes notificacions es queden a la barra superior de l'Android o barra d'estat. Les notificacions poden tenir definida una prioritat, que dona una pista a la interfície gràfica sobre com mostrar-les. Per a crear aquestes notificacions hem d'utilitzar les classes *Notification* i *NotificationManager*.

Estils i temes

En l'Android es poden definir estils i aquestes propietats són heretades pels elements fills de l'objecte visual on ho hem definit.

Els estils són una col·lecció de propietats que defineixen el format i la visualització d'una finestra o objecte *View*. Aquestes propietats són la mida, el color de fons, l'espai exterior, l'espai interior, etc. Tots estan definits en un fitxer XML, separat del que serveix per a definir el *layout* de les nostres activitats.

Aquests estils es podrien definir de manera explicita dins de les definicions mateixes de la interfície (fitxers XML del directori *layout*), però separar-los millora la separació de conceptes i la reutilització. És semblant als estils CSS[3] que serveixen per a definir l'estil d'una pàgina web.

[3] fulls d'estil en cascada o *cascading style sheet*

Aquests estils es defineixen amb la propietat *style* d'un element *View*. Es poden definir al fitxer l'XML dins de la definició de l'objecte *View* o també en els constructors d'aquests objectes *View*.

Un estil pot heretar d'un altre; en cas de ser un de definit per nosaltres simplement hem de donar el nom del nostre estil prefixat a l'estil que volem heretar.

Exemple

Tenim un estil *A* que defineix el tipus de lletra. Podem tenir un estil *B* que defineix el color:

```
<style name="A.B">
      <item name="android:textColor">#FFFF00</item>
</style>
```

Si volguéssim un estil *C* que hereta de *B* mantenint *A*, es cridaria a *A.B.C* i així successivament.

En el cas d'heretar un estil natiu de l'Android hem de definir l'atribut del fitxer XML *parent*, com per exemple *parent="@android:style/TextAppearance"*.

Un tema és un estil aplicat a tota una activitat o aplicació, en lloc d'una vista en concret. Per exemple, podem definir un estil de mida de lletra i una activitat i tots els textos dins d'aquesta activitat tindran aquesta mida de lletra.

La potència dels temes és que permeten ser canviats fàcilment i canviar la visualització de la nostra aplicació per complet. Aquests temes s'apliquen per mitjà de l'atribut *android:theme* d'una activitat.

Objectes 3D i animacions

Per a donar més potencial a la nostra aplicació es poden fer animacions i objectes amb textures de tres dimensions o 3D.

En el cas de 3D ens podem basar en la biblioteca OpenGL, en concret l'OpenGL ES API, que permet fer renderitzatges amb alt rendiment de tres dimensions. Per a això es basa en biblioteques on simplement s'ha d'escriure el nostre objecte *View* corresponent i implementar el mètode *onDraw* per a escriure l'objecte 3D que vulguem.

També podem tenir objectes 3D utilitzant el sistema *Renderscript* per a maximitzar el rendiment, ja que es tracta d'una biblioteca de baix nivell que executa codi nadiu, en concret codi C. Aquest sistema, a més, evita haver de preocupar-nos de les diferències de baix nivell dels dispositius, i això ho fa per mitjà de la compilació en el moment (*just in time*) des del *bytecode* fins al codi màquina. Aquest tipus de codificació maximitza el rendiment, però fa més complicats el desenvolupament i la depuració del codi.

Les animacions serveixen per a definir transicions dels nostres objectes visuals. Aquestes animacions poden ser definides dins dels nostres fitxers XML, en el directori *anim*, o bé per mitjà del codi mateix.

Exemple d'animació

Una animació pot ser el canvi d'una imatge per altres, o la rotació o la desaparició d'un botó.

Altres maneres de definir la interfície

Hem vist que una manera típica de definir els nostres elements visuals és per mitjà dels fitxers XML de recursos corresponents al contenidor. Aquests fitxers permeten triar la majoria d'atributs, i permeten ser accessibles des del codi mateix. És bastant habitual tenir la definició inicial de la nostra interfície gràfica en un fitxer XML i després modificar-la per mitjà del codi, accedint als elements per mitjà de la classe *R*.

Però no tots els elements gràfics que hi ha en les definicions de l'XML s'acaben renderitzant: només aquells que són en la línia d'herència d'algun element renderitzat. Per això en la nostra activitat s'ha de cridar al mètode *setContentView* per a indicar quin és el contenidor de la nostra activitat. Això, a més, permet definir elements com recursos, però no renderitzar-los immediatament, però sí poder-los renderitzar *a posteriori*. Aquest procés és el conegut com a *inflate*. El procés utilitza l'objecte XML processat i genera la classe visual corresponent.

3.2. Esdeveniments (*events*)

Els esdeveniments (*events*) són molt semblants a altres entorns de desenvolupament d'interfície gràfica basada en esdeveniments. Aquests esdeveniments reaccionen a qualsevol acció de l'usuari, sia prémer una tecla al teclat físic o

virtual, prémer un botó físic, des del botó de menú fins al botó de la càmera, tocar un punt a la pantalla o fer accions més sofisticades (apropar o allunyar, moure objectes, etc.).

Aquests esdeveniments sempre es duen a terme, i el fil de la interfície els interpreta. En cas de voler fer una acció davant d'un d'aquests esdeveniments, haurem de registrar un escoltador o *listener* per a aquest esdeveniment en l'objecte visual que ens interessi. Segons el tipus d'esdeveniment provocat, aquest esdeveniment ens portarà més informació, que arribarà en forma de paràmetres a l'escoltador.

Exemple d'escoltadors

A continuació veiem l'ús típic dels escoltadors per a escoltar l'acció de clicar.

```
public class HelloWorldActivity extends Activity {
      @Override
      public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      Button btn = (Button) findViewById(R.id.button);
      btn.setOnClickListener(new View.OnClickListener() {
          public void onClick(View v) {
               Toast.makeText(HelloWorldActivity.this, "Click",
Toast.LENGTH_SHORT).show();
          }
     });

    }
}
```

El nostre codi no sempre serà l'únic interessat a escoltar aquests esdeveniments, i per tant, s'ha d'escriure un codi segur per a evitar problemes a altres components. Uns quants esdeveniments són d'una única consumició, és a dir, que no es poden tenir diversos escoltadors sinó que el primer que el rep fa que la resta no ho vegi.

Ara veurem uns casos d'esdeveniments més sofisticats i molt particulars de les plataformes de dispositius mòbils: esdeveniments tàctils i esdeveniments de gestos predefinits.

En el cas dels esdeveniments tàctils o *touch events*, hi ha dos grups, els esdeveniments d'una única acció i els esdeveniments multiacció o *multitouch events*.

Dels esdeveniments d'una única acció podem tenir:

- **onClick**. És un clic bàsic sobre un objecte visual.

- **onLongClick**. És un esdeveniment de clic, sense moviment d'aquest, però amb una durada més elevada.

- **onDown**. És quan un usuari polsa la pantalla. És a dir, en el moment de polsar, seria l'equivalent al *keyDown* d'una tecla o botó.

- **onScroll**. Es tracta de l'esdeveniment de fer desplaçament, tant vertical com horitzontal.

- **onSingleTapUp**. Succeeix quan s'allibera la pantalla polsada. En cas de fer-ho amb el dit, és quan es deixa de tocar la pantalla després d'una acció tàctil.

- **onFling**. És una successió d'esdeveniments, de tocar la pantalla, moure, i deixar de tocar la pantalla. Se sol utilitzar per a desplaçar objectes visuals.

Per al cas dels esdeveniments multiacció tenim diferents accions per a l'esdeveniment *onTouchEvent*[4], que es poden dividir en les següents:

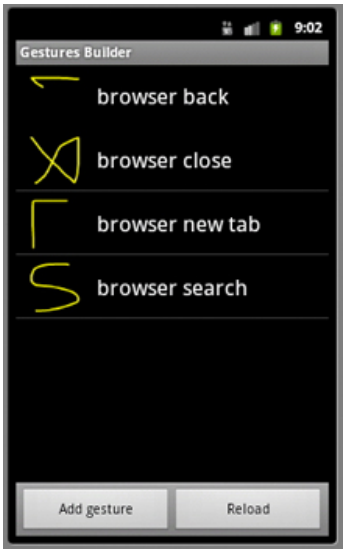
(4)L'esdeveniment *touch* és llançat a cada acció sobre la pantalla tàctil, incloent-hi les vistes anteriorment d'un únic toc i les actuals de múltiples tocs simultanis.

- *ACTION_DOWN*. Pulsació de la primera acció de tocar la pantalla.
- *ACTION_POINTER_DOWN*. Pulsació de la segona acció de tocar la pantalla.
- *ACTION_MOVE*. Acció de moviment mentre es tenen els dos punts polsats.
- *ACTION_UP*. Acció d'alliberar la pantalla corresponent a la primera pulsació.
- *ACTION_POINTER_UP*. Acció d'alliberar la pantalla corresponent a la segona pulsació.

Aquestes accions poden ser usades, per exemple, per a fer un zoom.

A més d'aquestes accions complexes i molt personalitzables l'Android ens ofereix unes biblioteques per a gestionar els gestos predefinits. És a dir, quan volem associar accions a gestos definits que tenen sentit per a la nostra aplicació; això s'usa molt per a tenir dreceres a accions habituals, com per exemple en navegadors per a accions com obrir una nova finestra o anar cap enrere.

Per a definir aquest tipus d'accions l'Android ens aporta una eina per a ajudar-nos a construir-los, que és el Gesture Builder, i un esdeveniment per a escoltar quan produeix aquestes accions l'usuari.



Exemple de definició de gestos usant el Gesture Builder

L'eina la tenim com un programa d'exemple de cada SDK i la funció que té és la de registrar els gestos o *gestures* i les traduccions que volem que tinguin aquests gestos. Aquests gestos es desen en el nostre emulador en un directori concret per a poder utilitzar-los *a posteriori*.

Per a utilitzar els gestos creats els hem de col·locar en el nostre projecte, accessible des dels nostres recursos per a ser referenciat des del codi. Perquè el nostre codi sigui capaç d'escoltar aquests gestos, ho hem d'indicar a la nostra activitat, i la manera és afegint una zona amb una capa del tipus *GestureOverlayView*, que serà on escoltarem aquest tipus de gestos. En el mateix codi hem d'afegir un escoltador, el qual estarà escoltant sobre la capa abans definida i el tipus de gestos que hi ha construïts amb el Gesture Builder.

Exemple de preparació dels gestos

En aquest tros de codi veiem les parts més interessants de la preparació dels gestos.

```
public class ListenGesturesActivity extends Activity implements
OnGesturePerformedListener {
   private GestureLibrary mLibrary;

   ...
   public void onCreate(Bundle savedInstanceState) {
      ...
          mLibrary = GestureLibraries.fromRawResource(this,
          R.raw.numbers);

          if (!mLibrary.load()) {
          finish();
          }
          GestureOverlayView gestures =(GestureOverlayView)
          findViewById(R.id.gestures);
          gestures.addOnGesturePerformedListener(this);
          ...
   }
}
```

I l'ús per a predir quin gest s'ha de fer implementant l'esdeveniment *gesturePerformed*. Com veiem, ens dóna una llista de possibles esdeveniments, que anomena *prediccions*. Amb això ens permet triar depenent de quina precisió volem tenir.

```
public void onGesturePerformed(GestureOverlayView
           overlay,Gesture gesture) {
           ArrayList<Prediction> predictions =
           mLibrary.recognize(gesture);
           ...
           Prediction p = predictions.get(i);
          String text = "got " + p.name + " with a precision of " +
           p.score;
           ...
}
```

3.2.1. *AsyncTask* i carregadors (*loaders*)

Sempre que es fa una acció que pugui bloquejar la interfície s'ha de fer en un fil d'execució diferent. A més, en el cas de l'Android és preocupant, ja que pot aparèixer un diàleg modal informant que l'aplicació no està responent, si triga més de 5 s a fer-ho. Això es pot fer com sempre en Java utilitzant les classes pròpies. Podria ser una mica així:

```
public void onClick(View v) {
   new Thread(new Runnable() {
      public void run() {
          Bitmap b = loadImageFromNetwork();
          mImageView.setImageBitmap(b);
      }
   }).start();
}
```

Aquest codi té un problema afegit, i és que s'està manipulant la interfície des de fora del fil de la interfície (*UI thread*), que no està preparat per a ser modificat per múltiples fils, amb els problemes que això pot comportar.

Per a evitar aquest problema l'Android ofereix diverses maneres de modificar l'UI[5], com són *Activity.runOnUiThread(Runnable)*, *View.post(Runnable)*, *View.postDelay(Runnable, long)* o *Handler*. Però aquestes classes fan el codi poc elegant i mantenible.

(5) interfície gràfica o interfície de comunicació amb l'usuari, o *user interface*

També es podria fer utilitzant una classe específicament creada per a les actualitzacions asíncrones de la interfície anomenada *AsyncTask*. En aquest cas quedaria com segueix:

```
public void onClick(View v) {
   new DownloadImageTask().execute("http://example.com/image.png");
}


private class DownloadImageTask extends AsyncTask<String, Void, Bitmap> {
      protected Bitmap doInBackground(String... urls) {
          return loadImageFromNetwork(urls[0]);
      }
      protected void onPostExecute(Bitmap result) {
          mImageView.setImageBitmap(result);
      }
}
```

Amb aquesta classe es poden fer totes les modificacions de la interfície de manera segura per a la interfície. Com veiem, *AsyncTask* utilitza els *Generics* de Java per a tenir un codi més controlat i elegant, i a més permet al programador afegir la lògica sobreescrivint els mètodes següents:

- **onPreExecute**. Invocat dins del fil de la interfície, just abans de començar la tasca. Útil per a actualitzar la interfície gràfica, per exemple amb una barra de progrés.

- **doInBackground(Params...)**. Invocat en un fil diferent després de *onPreExecute*, és on es fa el treball costós. En aquest mètode es poden fer *publishProgress(Progress...)* per a informar la interfície del progrés.

- **onProgressUpdate(Progress...)**. Invocat al fil de la interfície després de cada crida al *publishProgress(Progress...)* fet al pas anterior.

- **onPostExecute(Result)**. S'executa al fil d'execució de la interfície en finalitzar el treball.

Aquestes *AsyncTask* poden ser cancel·lades en qualsevol moment, la qual cosa acaba cridant a un altre mètode per a poder cancel·lar correctament la nostra *AsyncTask* i alliberar els recursos.

Un altre element important són els carregadors o *loaders*. Aquests serveixen per a carregar de manera asíncrona dades en la nostra aplicació.

Les característiques principals dels carregadors són:

- Poden ser associats a cadascuna de les activitats o fragments.

- Permeten recollir dades de manera asíncrona.

- Monitoren els recursos, de tal manera que quan hi ha canvis en les dades automàticament lliuren aquestes dades.

- Després d'un canvi de la configuració, es connecten automàticament cursor de les dades abans del canvi. Així no és necessari tornar a preguntar per les dades esmentades.

És responsabilitat del desenvolupador encarregar-se d'aconseguir les dades de manera correcta i mostrar-les a l'usuari, i els carregadors s'encarregaran de mantenir les dades i la nova informació.

4. Altres parts de l'SDK

L'Android té moltes parts, i cadascuna és realment gran, i per això es fa difícil cobrir-ho tot i segurament moltes d'aquestes parts no seran necessàries per al desenvolupament de la nostra aplicació. Per això ara passarem a veure algunes d'aquestes parts que són importants, parts relacionades amb biblioteques que existeixen i que ofereixen des d'emmagatzemament d'informació fins a accés a sensors o millores en la usabilitat, o l'acompliment de la nostra aplicació.

4.1. Arxivament de dades

Es poden emmagatzemar dades de diverses maneres:

- Per mitjà de les preferències de l'aplicació.
- En fitxers en l'emmagatzemament intern.
- En fitxers en l'emmagatzemament extern.
- Per mitjà de bases de dades.
- Per mitjà de la Xarxa.

En el cas d'emmagatzemar dades en les preferències de l'aplicació, s'utilitza la classe *SharedPreferences*, per mitjà dels mètodes de l'activitat *getSharedPreferences()* o *getPreferences()*. Aquí podrem guardar parells de clau-valor, en què el valor seran tipus primitius. Aquestes dades seran automàticament persistides entre diferents sessions d'usuari.

L'emmagatzemament intern i extern es refereix a desar fitxers en la memòria interna del dispositiu o bé en la memòria externa, generalment una targeta de memòria. Hi ha mètodes per a cada cas, de manera que es pot obrir un fitxer amb el nom concret, llegir-lo i escriure'l i finalment tancar-lo. També hi ha la possibilitat de treballar amb els fitxers del cau, el propòsit dels quals és millorar l'experiència d'usuari evitant haver de fer accions innecessàries.

> És important saber que hi ha alguns directoris en els quals els fitxers desats seran visibles per a totes les aplicacions i no s'esborraran encara que es desinstal·li l'aplicació; per exemple, tenim el directori *Music*, *Podcasts*, *Ringtones*, etc.

En el cas de l'emmagatzemament en base de dades s'utilitzarà SQLite. Les bases de dades que creem seran accessibles per nom des de qualsevol classe de la nostra aplicació però no des de fora d'aquesta. És recomanable utilitzar la classe *SQLiteOpenHelper* per a crear les nostres bases de dades i treballar-hi.

Sobre aquestes bases de dades es podran fer tot tipus de sentències SQL. També tindrem l'opció de recuperar dades en un cursor, i treballar amb aquestes.

Finalment per a desar dades a la Xarxa, s'ha de fer accedint a la Xarxa i enviant la informació per mitjà de serveis en línia.

4.2. Accés natiu

L'Android Native Development Kit (Android NDK) és un conjunt d'eines que ens permet encastar components que usen codi natiu en les nostres aplicacions d'Android tradicionals.

Aquestes parts de les nostres aplicacions s'escriuran en C i C++, i aprofitaran que són aplicacions natives per a reaprofitar certes parts o aconseguir més rendiment.

Aquest tipus d'aplicacions tenen un increment de la complexitat, i només per la naturalesa que tenen no aconseguiran millores, sinó que haurem de saber exactament què hem de fer per a aprofitar millor els recursos de baix nivell. Per això és important saber quan cal utilitzar-ho.

> Una bona regla general és utilitzar l'NDK només quan sigui necessari, no sense haver examinat abans altres alternatives.

Les aplicacions candidates a utilitzar l'NDK solen tenir un ús intensiu de CPU però sense utilitzar gaire memòria, com per exemple aplicacions de processament de senyals de ràdio o simulacions de físiques.

Hi ha dues maneres d'afegir els nostres components NDK a una aplicació d'Android:

- Implementant la nostra aplicació de manera tradicional, amb les API de l'SDK de l'Android i utilitzar JNI per a fer crides als nostres components NDK. És disponible des de la versió 1.5 de l'Android.

- Implementant les classes directament natives, utilitzant la classe *NativeActivity*. La seva definició canvia en el manifest, però el cicle de vida no. És disponible des de la versió 2.3 de l'Android.

JNI

El Java Native Interface (JNI) és un entorn de programació que permet que un programa escrit en Java i executat en una màquina virtual pugui accedir a programes escrits en altres llenguatges com C, C++ o Python.

Dins de l'NDK s'inclouen una sèrie d'eines com compiladors o assembladors, per a generar binaris per a l'arquitectura ARM en diferents plataformes de desenvolupament com Linux, Windows o OS X.

4.3. Serveis basats en localització (*location based services*)

Els serveis basats en localització o aplicacions basades en localització (*location based services* o *location based application*, LBS) són aplicacions o serveis basats en la posició geogràfica de l'usuari.

Per a determinar la posició geogràfica d'un dispositiu mòbil en la nostra aplicació, es pot usar GPS o bé la localització basada en les cel·les de telefonia i els senyals Wi-Fi. Aquestes possibilitats es coneixen com a *LocationProvider*. Les aplicacions que volen conèixer la geoposició es poden basar en només una de les alternatives o en totes dues.

El sistema de posicionament global o *global positioning system* (GPS) és un sistema per a determinar la posició d'un objecte, persona o vehicle a tot el món, i pot arribar a precisions de centímetres encara que normalment parlem de metres.

Desenvolupament del GPS

El sistema GPS va ser desenvolupat i instal·lat pel Departament de Defensa dels Estats Units. Per a poder donar aquest servei hi ha desenes de satèl·lits al voltant de la Terra a una distància de més de 20.000 km.

Hi ha altres alternatives molt similars desenvolupades per altres potències mundials, com són el GLONASS de la Federació Russa o el Galileo de la Unió Europea.

El GPS és més precís, però té uns inconvenients en què les localitzacions via Wi-Fi o cel·la de telefonia tenen avantatge. Amb el GPS el consum de bateria és més gran i el temps per a aconseguir la posició és més llarg.

La precisió de les posicions, i també el consum de bateria que hi ha associat, pot quedar afectat per factors com el moviment de l'usuari, el canvi de proveïdor de localització o els canvis de la precisió del senyal. Tot això ho ha de tenir en compte la nostra aplicació i actuar en conseqüència per a evitar al màxim l'ús inapropiat del servei.

Per a poder conèixer la localització en l'Android la manera de fer-ho és subscrivint un *LocationListener* als esdeveniments relacionats amb la localització. Els objectes que implementen la interfície *LocationListener* han d'implementar els mètodes *onLocationChanged*, *onProviderDisabled*, *onProviderEnabled* i *onStatusChanged*. Aquests serveixen per a escoltar canvis en l'estat i en la disponibilitat dels proveïdors de localització en què estem interessats.

Aquest objecte que serveix per a rebre les actualitzacions s'ha de registrar al sistema per mitjà del mètode *requestLocationUpdates*, en què especifiquem el proveïdor que ens interessa (*GPS_PROVIDER* o *NETWORK_PROVIDER* per a la

localització basada en cel·les i xarxes Wi-Fi), i també temps i distància entre actualitzacions; finalment afegim el nostre objecte per a rebre les actualitzacions.

Un mètode interessant és poder conèixer l'última localització coneguda: *getLastKnownLocation*. Ens permet evitar haver d'esperar els proveïdors de localització i possiblement estalviar bateria, però amb la pèrdua de precisió corresponent.

En totes les localitzacions de què parlem (que pertanyen a la classe *Location*) podrem saber quan van ser capturades aquestes localitzacions i per mitjà de quin proveïdor, i així la nostra lògica podrà ser capaç de decidir si és suficient aquesta localització o en volem una de nova.

A més de poder interactuar directament amb els proveïdors de localització, hi ha la possibilitat d'utilitzar biblioteques de propietat, com és l'API de Google Maps proporcionada per Google. Aquesta biblioteca proporciona un *ViewGroup* anomenat *MapView*, que es mostra com els mapes propis de Google Maps, utilitzant les dades del seu servei, i també la seva interfície d'usuari. Inclou la gestió de la posició i les capes pròpies d'aquesta aplicació. Sobre això es pot programar per a afegir noves capes amb la nostra informació, però per a poder fer això és imprescindible obtenir una clau vàlida.

4.4. Comunicacions

Per a dur a terme comunicacions hi ha diferents opcions en l'Android. Hi ha la possibilitat de dur a terme comunicacions de xarxa normals, per mitjà de l'API de comunicació de xarxa, o controlar l'estat dels nostres proveïdors de xarxa amb el mètode *Context.getSystemService(Contex.CONNECTIVITY_SERVICE)*. Però també hi ha la possibilitat d'utilitzar API per a propòsits concrets. Podem destacar NFC, Bluetooth i SIP.

Com sempre, per a cadascuna d'aquestes connexions es fa necessari especificar en el nostre manifest els permisos adequats, i en cas de no fer-ho la nostra aplicació no funcionarà.

4.4.1. Bluetooth

En l'Android, programant per treballar amb Bluetooth es pot:

- Escanejar per trobar altres dispositius.
- Demanar a l'adaptador de Bluetooth local per dispositius ja enllaçats.
- Establir canals RFCOMM.
- Connectar amb altres dispositius per mitjà del servei de *discovery*.
- Transferir dades des d'altres dispositius i a altres dispositius.
- Gestionar múltiples connexions.

Per a treballar amb Bluetooth i aconseguir les accions bàsiques, com són activar Bluetooth, trobar dispositius ja enllaçats o nous, i transferir dades, l'Android ens proporciona una sèrie de classes clau:

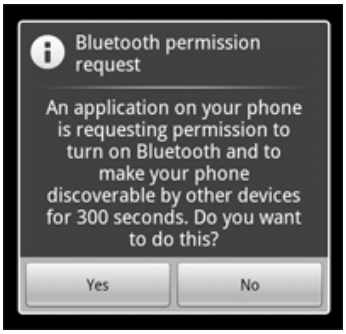
- **BluetoothAdapter**. És el principal punt d'accés en la interacció. Ens serveix per exemple per a trobar dispositius ja enllaçats o no, i crear sòcols per a comunicacions amb altres dispositius.

- **BluetoothDevice**. Representa un dispositiu, que podem interrogar sobre informació o bé obrir connexions contra aquest.

- **BluetoothSocket**. És el canal de comunicacions per a intercanviar informació.

- **BluetoothServerSocket**. Serveix per a rebre connexions quan el nostre dispositiu està en mode servidor.

En moltes de les interaccions de la nostra aplicació amb Bluetooth l'Android mostrarà diàlegs d'informació per a confirmar les accions.



Imatge de missatge de l'Android que avisa d'accions provocades per la nostra aplicació
Font: http://developer.android.com

4.4.2. NFC

NFC, a diferència d'altres tecnologies de comunicació via radiofreqüències com Bluetooth o Wi-Fi, permet fer les connexions entre dos dispositius sense necessitat de descobrir aquests dispositius ni enllaçar-los, i les interaccions es poden iniciar només amb una acció.

NFC serveix perquè el nostre dispositiu sigui capaç d'interactuar amb *tags* NFC. El nostre dispositiu podrà llegir la informació d'aquests *tags* o fins i tot interactuar. Aquests *tags* tenen diferents tecnologies i poden ser des de molt senzills amb informació textual fins a molt complexos amb capacitats de càlculs o xifratge; fins i tot un dispositiu Android pot emular que és un *tag*.

Per a poder treballar amb NFC l'Android ens proporciona les classes següents:

- **NfcManager**. Serveix per a gestionar diversos adaptadors físics de dispositius nostres a escala NFC. Com que en la majoria de casos només en tenim un, es pot usar el mètode estàtic *getDefaultAdapter*.

- **NfcAdapter**. Representa el nostre adaptador al dispositiu NFC. Serveix per a registrar canvis de *tag* cap a la nostra activitat, i treballar amb missatges NDEF.

NDEF

NDEF és una estructura de dades definida per l'NFC Forum per a emmagatzemar de manera eficient dades en els *tags* NFC.

- **NfcMessage** i **NfcRecord**. *NfcMessage* és un contenidor de zero o més *NfcRecords*. Cada *NfcRecord* té informació NDEF.

- **Tag**. Representa l'objecte passiu d'NFC. Aquests *tags* poden tenir diferents tecnologies i es poden preguntar mitjançant el mètode *getTechList()*.

4.4.3. SIP

El protocol d'inici de sessió o *session initiation protocol* (SIP) és un protocol per a controlar les sessions de comunicacions multimèdia, com per exemple de trucades de veu o de vídeo sobre IP. El protocol es pot usar per a crear, modificar i finalitzar sessions entre dos punts (unidestinació) o entre múltiples punts (multidestinació). En les modificacions es poden canviar el nombre de punts implicats o modificar els *streamings*.

En l'Android hi ha una sèrie de classes que ajuden a treballar amb aquest protocol i, per tant, facilitar el desenvolupament. Això ens permetrà fer videoconferències en les nostres aplicacions i també missatgeria instantània.

Cal tenir en compte alguns requisits i limitacions que hi ha en el desenvolupament de SIP per a l'Android:

- Només és disponible per a dispositius 2.3 o superiors.
- SIP funciona amb una xarxa de dades sense fils, i per tant no es pot provar per mitjà dels emuladors, només amb dispositius físics.
- Cada membre de la comunicació necessita un compte SIP, identificat en un servidor SIP.

Hi ha una sèrie de classes clau que són les encarregades principals de treballar amb el protocol:

- **SipManager**. Serveix per a treballar amb les tasques SIP, com és iniciar connexions, o donar accés a serveis SIP.
- **SipProfile**. Defineix un perfil SIP, incloent-hi la informació del servidor i del compte.
- **SipAudioCall**. Serveix per a gestionar una trucada sobre IP per mitjà de SIP.

4.4.4. *Widgets*

Els *widgets* són aplicacions en miniatura que permeten mostrar parts d'una aplicació dins de l'escriptori o pantalla d'inici.

Exemple de *widgets* a la pantalla d'inici d'una tauleta tàctil i d'un telèfon intel·ligent

Es pot programar la cerca d'actualitzacions dels *widgets*, que es du a terme per mitjà d'un tipus especial de receptor d'esdeveniments (*broadcast receiver*), amb la periodicitat que es cregui convenient.

Hi ha tres elements bàsics per a la construcció d'un *widget*:

- *AppWidgetProviderInfo*. Serveix per a definir, generalment per mitjà de l'Android Manifest, la informació de configuració del *widget*. Entre aquesta informació destaca la grandària mínima (alçària i amplària), el període d'actualització, el *layout* visual (definit de la mateixa manera que en el cas de les activitats o *activities*) i l'opció d'afegir una *activity* només dedicada a configurar-lo.

- *AppWidgetProvider*. L'encarregat d'actualitzar el *widget* segons els esdeveniments que vagin arribant. Esdeveniments importants que ha de gestionar són els d'actualització (*onUpdate*), activació (*onEnabled*), i esborra (*onDelete*). El més important és el d'actualització que, normalment, es fa des d'un *service* per a evitar bloquejos i el consegüent missatge d'error d'una aplicació que triga molt a executar-se (ANR: *application not responding*).

- El *layout* propi, que ha estat configurat anteriorment en el primer punt. Es defineix de la mateixa manera que el cas de les activitats: utilitzant un fitxer XML i les vistes (*views*) tradicionals.

Atenció

Hem de ser acurats quan programem les cerques d'actualitzacions perquè els *widgets* també s'actualitzen mentre el dispositiu està apagat, la qual cosa pot consumir molta bateria.

Podeu obtenir més informació sobre els *widgets* a la pàgina següent:

http://developer.android.com/guide/topics/appwidgets/index.html

4.4.5. Sincronització de dades

L'Android permet crear sistemes de sincronització de dades entre el nostre dispositiu i altres punts, com pot ser servidors propis o servidors en el núvol.

Com a requisit per a poder sincronitzar dades, és imprescindible tenir un compte d'usuari de l'Android. Generalment, almenys se sol tenir el compte de Google, que ens permet sincronitzar contactes, calendaris, etc. A més, po-

dem crear els nostres propis serveis d'usuari per mitjà de l'*AccountManager* i l'*AbstractAccountAuthenticator*, la qual cosa ens donaria accés a comptes d'altres serveis.

El procés de sincronització pot ser llançat de manera automàtica pel sistema operatiu mateix, la qual cosa és ideal per a mantenir totes les aplicacions sincronitzades des d'un punt central. Per a això, haurem de desenvolupar un servei que s'encarregarà de cridar un adaptador per a la sincronització (*SyncAdapter*), que serà qui faci la lògica de la sincronització. Aquest servei de sincronització, per mitjà del *SyncAdapter*, haurà de treballar amb algun *ContentProvider*, per llegir les dades que s'han de sincronitzar i desar-les posteriorment.

Exemple

Un cas molt habitual és sincronitzar les dades dels nostres contactes amb alguna xarxa social i afegir actualitzacions d'aquests contactes a les xarxes socials.

Per a més informació, tenim una mostra d'implementació disponible com a part dels exemples de l'SDK.

5. Eines de desenvolupament de l'Android

Fins ara hem vist la potència de les nostres aplicacions, en què podem definir accés a pràcticament totes les parts del nostre dispositiu en la nostra aplicació, amb diverses maneres de fer la mateixa acció (per exemple, per mitjà de fitxers XML o en codi). Però moltes vegades s'ha de tenir en compte molta informació i fer-ho sense ajuda és difícil.

Per a això en aquest apartat veurem les principals eines de les quals disposem. Veurem les eines estàndard de l'SDK d'Android, algunes en profunditat però no totes, ja que a més d'escapar-se de l'abast d'aquest apartat se n'afegeixen constantment de noves.

Després veurem els entorns de desenvolupament principals, prestant especial atenció a l'entorn oficial de Google per a desenvolupament.

Finalment veurem algunes eines externes per a poder veure la potència de la nostra plataforma.

5.1. SDK d'Android

Algunes de les eines que vénen amb l'SDK[6] oficial d'Android són generals i s'usen per a cadascuna de les nostres aplicacions; en canvi, n'hi ha algunes que són molt especifiques del tipus de desenvolupament que vulguem fer, o les millors que tenim pensades sobre l'aplicació.

[6] equip de desenvolupament de programari o *software development kit*

Algunes de les més interessants poden ser:

- **sqlite3**. Eines per a gestionar bases de dades per mitjà de sentències SQL.
- **hprof-conv** i **dmtracedump**. Eines per a fer perfilament o *profiling* de la nostra aplicació, és a dir, per a millorar el rendiment trobant possibles problemes de memòria o CPU.
- **layoutopt** i **Draw 9-patch**. Eines per a millorar la nostra interfície gràfica. La primera per a analitzar el rendiment dels nostres *layouts* i així poder-ne optimitzar el rendiment. La segona per a la creació de gràfics.
- **monkey** i **monkeyrunner**. Eines per a fer tests sobre la nostra aplicació.
- **zipalign**, **ProGuard**. Eines per a millorar els fitxers corresponents a la nostra aplicació.

- **emulator**. L'emulador on podem provar les nostres aplicacions.
- **logcat**. Visualitzador dels registres (*logs*) de sistema d'un dispositiu o emulador.
- **android**. Gestor dels dispositius virtuals o AVD.
- **adb**. Eina per a poder veure l'estat del nostre emulador.

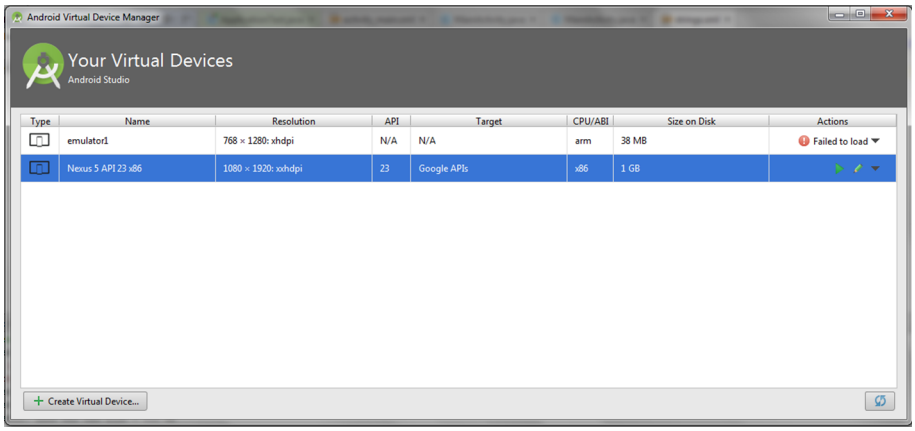
N'hi ha alguns que són usats constantment i que requereixen una mica més d'explicació.

Per a començar a desenvolupar el primer que s'ha de fer és crear un dispositiu virtual (AVD), que correspon a una configuració de l'emulador, definint les opcions de maquinari i programari que volem que tingui aquest emulador. Si no es disposa d'una eina més potent com l'AVD Manager s'ha de fer des de la línia de comandes, des d'on amb la instrucció *android* es poden enumerar, crear, esborrar o actualitzar els AVD disponibles. També ho podem fer des de l'AVD Manager, que ens permet fer el mateix, tal com es veu en les imatges següents.

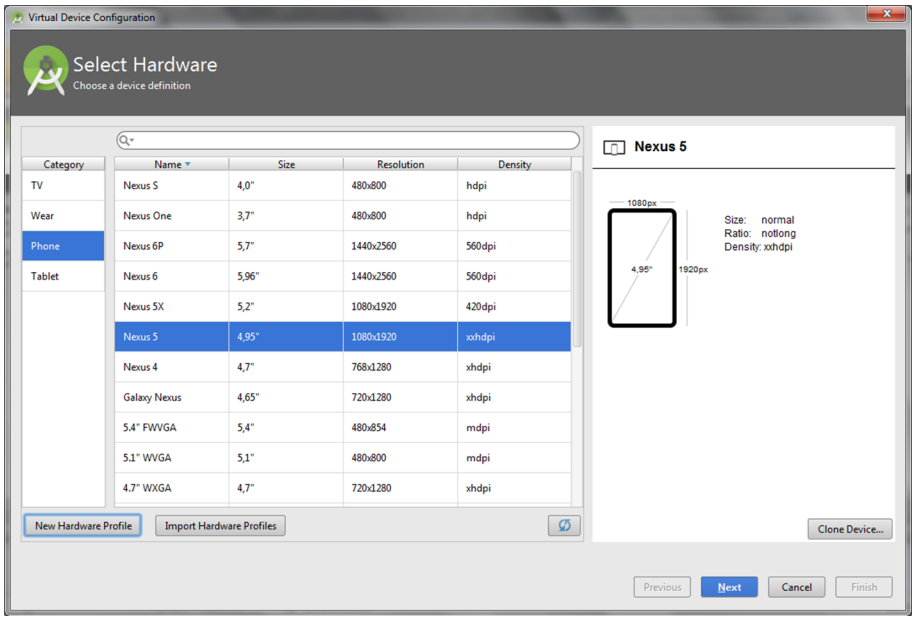
Ajuda de comandes *android*

Per a conèixer les opcions exactes de la comanda *android* podem veure l'ajuda amb *android-help*.

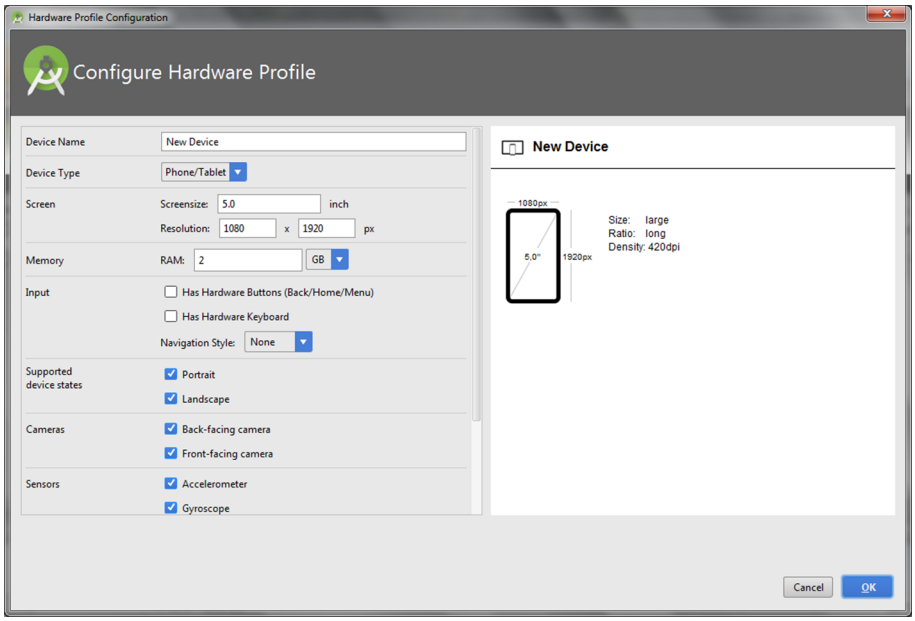
Vista de l'AVD Manager



Creació d'un nou AVD



Creació d'un nou AVD



A més, ens permet poder baixar noves versions de les biblioteques de l'SDK, de manera que podem actualitzar sempre des d'aquí abans de l'aparició d'una nova versió per a desenvolupament.

Una eina important és l'*adb* que s'instal·la en el nostre emulador o dispositiu, que permet veure informació d'estat de l'aplicació, fer bolcats d'estat per a analitzar-los, accedir als registres, i fins i tot depurar l'aplicació o tenir un entorn d'ordres de l'emulador.

És important poder tenir traçabilitat de les accions que van succeint, i per a això en el nostre codi podem utilitzar la classe *android.util.Log* amb els mètodes per a escriure sobre el registre segons la verbositat que vulguem. Per a poder veure aquesta informació hi podem accedir usant la subinstrucció *logcat* de l'ordre *adb*.

Sens dubte una de les eines essencials és l'emulador de l'Android, accessible amb l'ordre *android*. Amb aquest emulador podem emular moltes accions d'usuari, moltes directament amb combinacions de tecles, que poden fer de des d'accedir al menú o obrir la càmera fins a canviar l'orientació de la pantalla. També n'hi ha algunes altres que es poden fer des de la consola de l'emulador, on podem canviar la velocitat de la xarxa, simular SMS, canviar la posició geogràfica, etc.

La majoria d'aquestes eines són línies de comandes i requereixen una sèrie de paràmetres específics per a treballar-hi, a més de necessitar canviar de context constantment per a poder fer treballs habituals. Per a millorar això, moltes d'aquestes eines s'agrupen en els connectors o extensions de desenvolupament per a l'Android, dels quals es pot destacar el que té suport oficial de Google: *Android Studio*.

Android Studio

És l'IDE oficial per a desenvolupar aplicacions Android i està basat en IntelliJ IDEA de JetBrains.

Cal tenir en compte, però, que hi ha altres emuladors disponibles a part del que incorpora Android Studio, com pot ser Genymotion.

5.2. Android Studio

És l'IDE oficial per a desenvolupar en Android i es basa en IntelliJ.

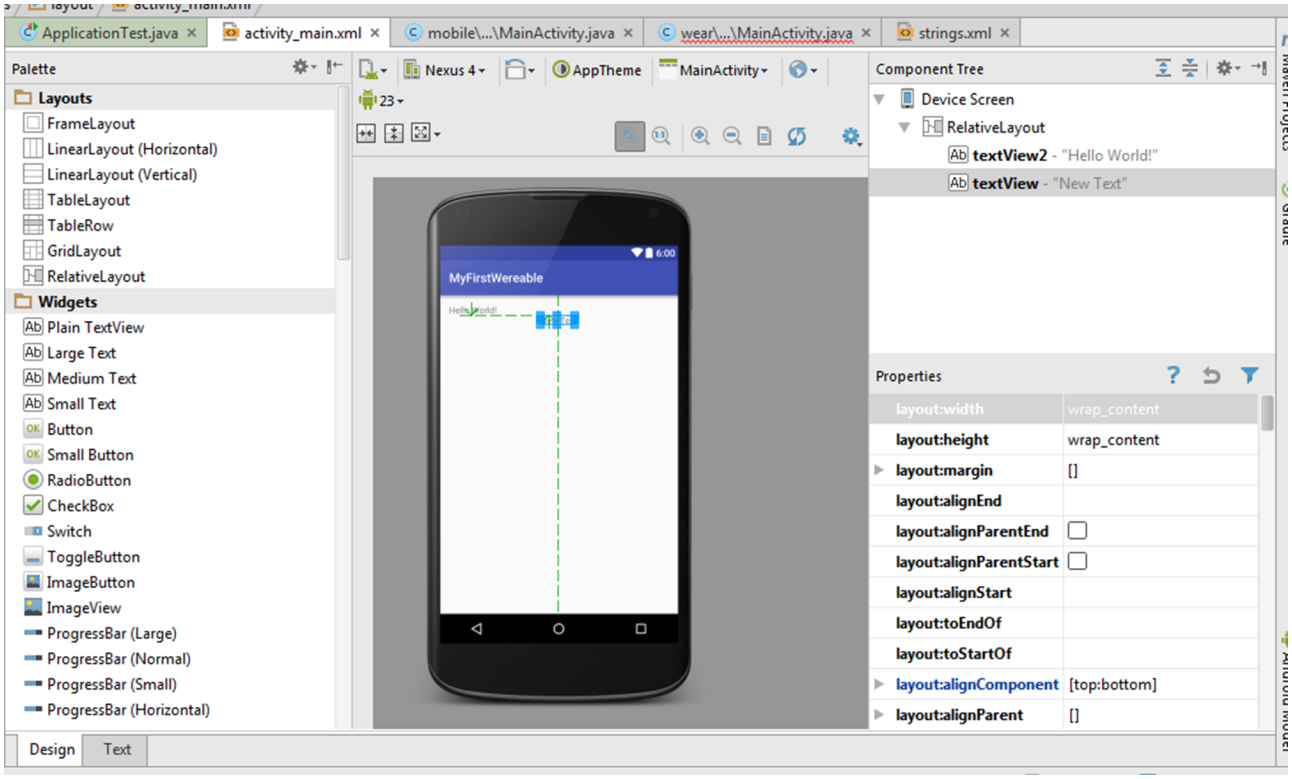
Android Studio incorpora les eines prèviament vistes i de manera transparent per al desenvolupador, integrades en una IDE. De manera que es pot fer des d'una depuració d'una aplicació en un dispositiu o emulador, fins a veure la informació de perfilament de la memòria o el sistema de fitxers actual i treballar-hi.

IDE

Acrònim d'«*Integrated Development Environment*», entorn integrat de desenvolupament que agrupa diferents eines per a la implementació i depuració d'aplicacions.

Entre altres coses que ens ofereix la interfície de desenvolupament d'Android hi ha la gestió integrada dels fitxers XML per mitjà d'editors gràfics. Una de les més interessants és l'edició gràfica dels les interfícies gràfiques, tal com es veu a la figura següent.

Edició d'interfície gràfica des de l'Android Studio



Aquesta edició gràfica, a més, porta incorporada la gestió de les diferents versions dels recursos alternatius, que depenen de cada característica, cosa que facilita molt el desenvolupament.

5.3. Depurant l'aplicació en dispositiu real

Malgrat que el nostre emulador és una eina potent, hi ha coses que no es poden verificar totalment, com són els esdeveniments de multiacció, o alguns sensors o connexions com són Bluetooth o sensors com l'acceleròmetre o similars. Hi ha algunes eines de la comunitat per a millorar aquesta falta, però sens dubte la millor seguretat la proporcionen les proves sobre el dispositiu real.

A més sempre és desitjable fer proves tan reals com sigui possible amb el dispositiu objectiu, per a poder apreciar també la responsitivitat de la nostra aplicació.

Per a poder depurar la nostra aplicació sobre un dispositiu real hem de fer el següent:

- Marcar la nostra aplicació com a depurable (*debuggable*). Per a això hem de triar l'atribut *android:debuggable* al valor *true*.

- En el nostre dispositiu, permetre que es connecti el depurador (*debugger*). Anem a la pantalla principal, premem *Settings*, i després *Developer options* > *USB debugging*. Pot ser que no estiguin habilitades les *developers options*,

ja que per defecte estan ocultes. Per fer que estiguin disponibles, cal anar a **Settings > About phone** i prémer **Build number** set cops. Quan retornis a la pantalla anterior, veuràs **Developer options**.

- Fer que el nostre sistema operatiu reconegui el dispositiu. Dependrà del sistema operatiu que estiguem usant per a desenvolupar.

Finalment es pot llançar l'aplicació o depurar com ho faríem normalment des de l'IDE i disposarem de la mateixa informació que en cas de tenir un emulador.

5.4. Eines de verificació

L'Android, igual que la resta de desenvolupaments mòbils, requereix moltes proves per a aconseguir un programari de qualitat, proves amb eines de verificació i proves en dispositius reals.

Hi ha algunes de les eines que estan integrades en l'SDK mateix, basades sobretot en JUnit, que és una biblioteca típica de verificació unitària per a Java.

5.4.1. Eines de verificació incloses en l'SDK

Les eines incloses per a la verificació en l'SDK de l'Android es poden dividir en les següents:

- Tests unitaris normals, basats en les biblioteques de JUnit, però sense utilitzar ni instanciar objectes especials de l'Android. Serveixen per a provar lògica d'aplicació, principalment.

- Tests unitaris utilitzant la infraestructura de l'Android.

- Tests unitaris d'estrès.

Els primers són execucions normals de tests unitaris, que es basen a provar que la lògica fa el que ha de fer i comprovar-ho amb *asserts* tradicionals de JUnit. És a dir, comproven que el resultat de l'execució és l'esperat o dins dels rangs esperats; en cas de ser cert continuen fins a acabar i per tant el test és vàlid, i en cas de no ser-ho el test passa a comptar com a fallit.

El segon cas té una especificitat més propera a l'Android, i permet coses com interceptar o provocar els esdeveniments del cicle de vida d'una activitat, i per això es poden provar coses com fetes per a una aplicació i veure que recupera correctament l'estat que tenia. Això ho fa gràcies a l'API *Instrumentation*.

En el cas dels tests dins de la infraestructura de l'Android, ens proporcionen moltes classes base per als nostres tests, com *ActivityTestCase*, *ServiceTestCase* o *ApplicationTestCase*. A més d'això hi ha molts objectes *Mock* que permeten

emular el comportament estàndard de la nostra aplicació, perquè sigui més fàcil verificar de manera aïllada l'aplicació. La classe que s'encarrega d'executar els tests és l'anomenada *InstrumentationTestRunner*, que permet interactuar de manera més directa sobre les classes de la nostra aplicació.

Finalment hi ha una eina que serveix per a verificar de manera automàtica i pseudoaleatòria la nostra aplicació. Aquesta eina és el *Monkey*, que té la metàfora d'una mona tocant el dispositiu. Aquesta eina llança esdeveniments de clic, esdeveniments tàctils o gestos, a més d'esdeveniments de sistema.

En aquest *Monkey* es poden definir les opcions següents:

- Nombre d'esdeveniments per llançar.
- Limitacions com només llançar esdeveniments cap a un paquet de codi.
- Tipus i freqüència dels esdeveniments.
- Opcions de depuració (*debugging*).

A partir d'aquí es poden llançar els tests i l'eina ens avisarà els casos següents:

- Si volem verificar només un paquet i s'intenta executar alguna part fora d'aquest paquet, l'aplicació bloqueja aquesta execució, amb les possibles conseqüències.
- Si l'aplicació té un problema o una excepció no controlada, l'eina pararà i reportarà l'error.
- Si l'aplicació provoca que es vegi un error del tipus «Application not responding», l'eina pararà i reportarà l'error.

Amb aquesta eina es pot comprovar que la nostra aplicació es comporta de manera correcta davant de situacions d'estrès.

5.4.2. Eines externes de verificació

A part d'aquestes eines hi ha algunes eines externes que ens ajuden a fer tests més específics; per exemple:

- **Roboelectric**: tests unitaris per a cada activitat d'Android.

- **Robotium**: test d'acceptació. És a dir, es llança l'aplicació i el nostre test va fent accions com si fos un usuari normal. Si aquest test funciona, això significa que la part verificada de la nostra aplicació està acceptada i funcionant.

Enllaços d'interès

Per a saber més sobre Roboelectric podeu visitar la pàgina web següent:

http://roboelectric.org

Per a saber més sobre Robotium podeu visitar la pàgina web següent:

https://github.com/RobotiumTech/robotium

El Roboelectric permet executar tests unitaris de la mateixa manera que en el cas dels tests d'activitats llançats des de les eines de l'Android, però sense necessitat de llançar un emulador. Això fa que els tests siguin molt més ràpids.

En canvi, el Robotium serveix per a fer proves automatitzades del tipus de caixa negra, és a dir, tests en què no sabem què hi ha dins de l'aplicació però podem provar com si fóssim un usuari i rebre resposta com a tal.

Aquest tipus de test el que intenta és llançar esdeveniments com si fóssim un usuari, sense necessitat de conèixer els elements (*spinners*, *buttons*, *textboxes*, etc.) que hi ha en el codi de les activitats; per tant es basa molt en el que l'usuari veu.

Exemple

Aquest exemple prova de fer canvis navegant pels menús; com veiem, no s'especifica el tipus d'objecte sobre el qual fem clic sinó el text que conté.

```
public class EditorTest extends
                  ActivityInstrumentationTestCase2<EditorActivity> {

...
   public void testPreferenceIsSaved() throws Exception {
                    solo.sendKey(Solo.MENU);
                    solo.clickOnText("More");
                    solo.clickOnText("Preferences");
                    solo.clickOnText("Edit File Extensions");
                    Assert.assertTrue(solo.searchText("rtf"));

                    solo.clickOnText("txt");
                    solo.clearEditText(2);
                    solo.enterText(2, "robotium");
                    solo.clickOnButton("Save");
                    solo.goBack();
                    solo.clickOnText("Edit File Extensions");
                    Assert.assertTrue(solo.searchText
                    ("application/robotium"));
   }
...
}
```

Aquest tipus de tests són molt potents perquè no requereixen el coneixement de l'aplicació com en la resta de casos, i a més estan provant tota l'aplicació al complet. Com a contrapartida tenen problemes com que deixen de funcionar amb petits canvis de la interfície, com un canvi de text, i que són lents de construir i d'executar.

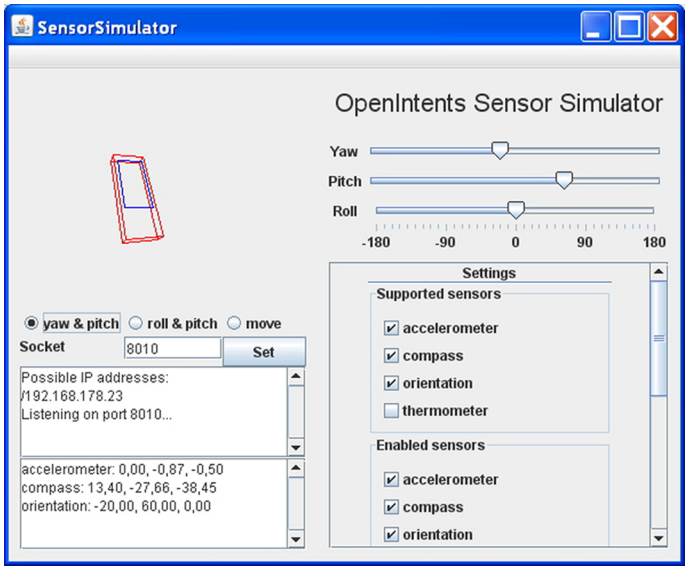
5.5. Altres eines

També hi ha altres eines generades per Google o per la comunitat. Entre aquestes destaquem App Inventor i Sensor Simulator.

5.5.1. Sensor Simulator

El Sensor Simulator és una eina creada per la comunitat per a poder simular els sensors del nostre emulador o dispositiu. L'objectiu és poder simular canvis d'orientació, o coses com canvis en la temperatura. Per a això es requereix un servidor corrent en el nostre entorn de desenvolupament i una aplicació instal·lada en l'emulador.

Ús del Sensor Simulator per a canviar dades dels sensors

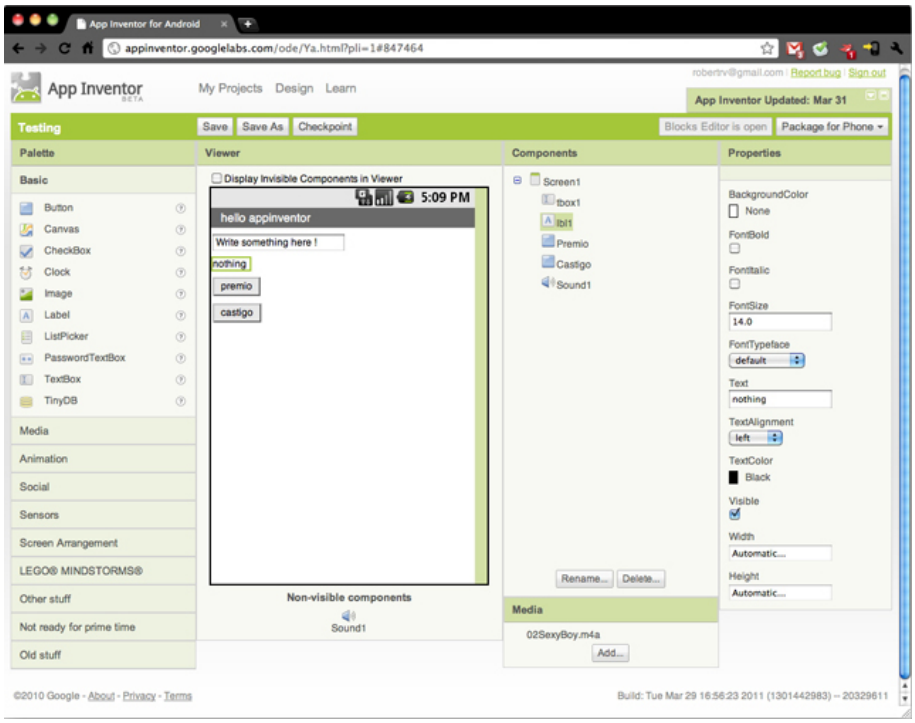


5.5.2. App Inventor

L'App Inventor és una aplicació en línia de Google el propòsit de la qual és que gent sense coneixements de programació sigui capaç de programar una aplicació senzilla i baixar-la i executar-la al seu mòbil. És un servei ubicat al núvol, per això totes les tasques es realitzen en el navegador. Per a fer-lo servir cal un compte Google.

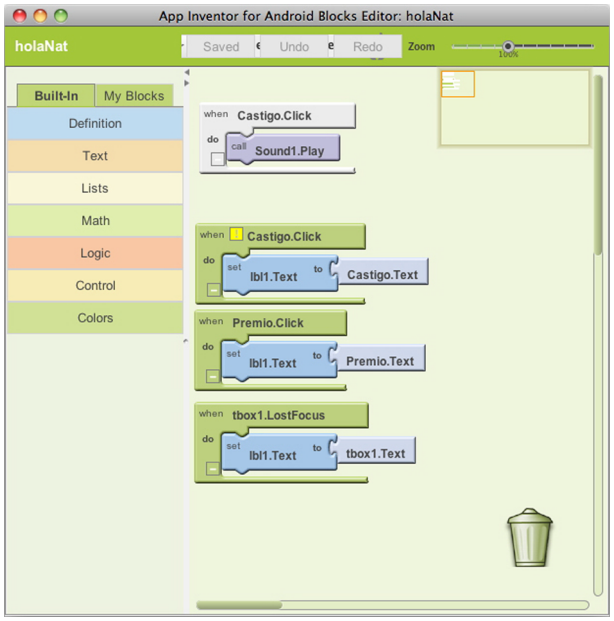
Per a facilitar l'edició visual ens incorpora un editor gràfic amb capacitats reduïdes i algunes peculiaritats per a fer-ne més senzill l'ús.

Exemple d'edició de la interfície gràfica amb l'App Inventor



A més, per a programar la lògica de l'aplicació utilitza objectes que representen les formes de control més típiques d'una aplicació mòbil:

Exemple d'edició d'una lògica d'aplicació utilitzant l'App Inventor



6. Distribució i negoci

Una vegada tenim acabada la nostra aplicació, o almenys una versió, és el moment de distribuir-la.

En el cas de l'Android es pot distribuir directament el fitxer *.apk* corresponent a l'aplicació, però també es pot fer per mitjà d'algun dels mercats d'aplicacions o *markets*, per exemple per mitjà del mercat oficial de Google, anomenat Google Play. En aquests mercats es pot oferir l'opció de vendre l'aplicació i aconseguir-hi remuneració, però sobretot obren el nou canal de distribució de la nostra aplicació.

Google Play

Google Play, com a aplicació de l'Android, només és disponible en els dispositius verificats per Google. Com que l'Android es pot utilitzar lliurement, segons la seva llicència, es pot usar en altres dispositius.

També hi ha altres mètodes de remuneració per les aplicacions, com poden ser per mitjà de la publicitat o bé per mitjà de pagaments en l'aplicació mateixa.

6.1. Signatura de l'aplicació

Signar una aplicació és un procés pel qual s'utilitza un certificat digital privat per a marcar l'aplicació, de manera que es pot saber unívocament si una persona és l'autora de l'aplicació. Aquesta unicitat es valida per mitjà de la clau pública corresponent.

Entitat certificadora

Una entitat certificadora (CA) emet certificats digitals verificant-ne l'origen i la validesa, com per exemple VeriSign o Thawte.

Per a poder utilitzar la nostra aplicació en un dispositiu físic, sempre ha d'estar signada, sia amb una clau definitiva o bé una de prova. El certificat, a diferència dels utilitzats en els navegadors, no requereix que sigui certificat per una entitat certificadora, sinó simplement pot ser generat pel desenvolupador mateix. Malgrat això, els certificats han de ser autèntics i vigents, encara que només se'n comprova la data de validesa la primera vegada que s'instal·la l'aplicació.

És important que la signatura de l'aplicació es faci amb un certificat que no caduqui durant la vida de l'aplicació; si succeeix això haurem d'assignar un nou paquet a la nostra aplicació per a poder signar-la correctament, i finalment l'usuari l'haurà d'instal·lar com una aplicació diferent. En concret, per a instal·lar l'aplicació per mitjà de Google Play, s'ha de signar l'aplicació amb un certificat amb final de validesa posterior al 22 d'octubre del 2033.

Els certificats es poden utilitzar per a signar més d'una aplicació; això pot ser per a diversos casos:

- Actualització de les *apps*: si volem que un usuari pugui instal·lar-se actualitzacions de l'*app*, hem de signar amb el mateix certificat durant tot el temps de vida de l'*app*. Si no, en detectar una signatura diferent, no instal·la l'*app*

actualitzada, s'hauria de renombrar o desinstal·lar manualment la versió prèvia.

- Dues aplicacions volen compartir informació en temps d'execució. Això és a causa que dues aplicacions amb el mateix certificat es poden executar dins del mateix procés, i per tant, tenen els mateixos recursos de sistema com si fossin un únic procés. Amb això es poden tenir aplicacions diferents que interactuen àmpliament.

- Modularitzar la nostra aplicació. Si tenim la nostra aplicació dividida en parts, i volem que cadascuna d'aquestes parts es pugui actualitzar separadament, una manera de fer-ho és fer aplicacions diferents signades amb el mateix certificat.

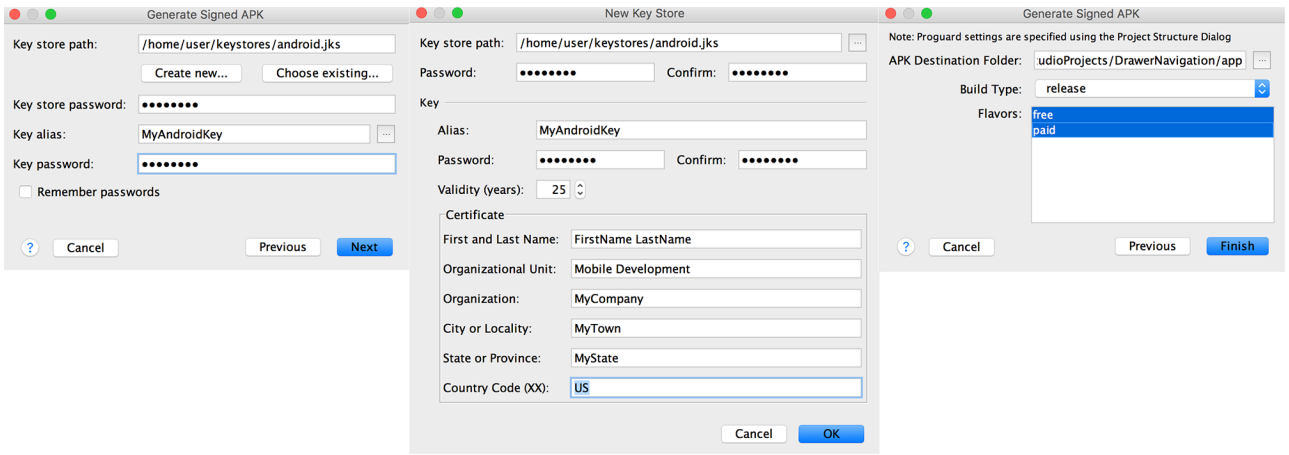
Les signatures de depuració es poden generar amb les eines estàndard d'Android Studio.

A l'hora de generar una versió pública hem de fer el següent:

1) Obtenir una clau privada.
2) Compilar l'aplicació per a ser distribuïda.
3) Signar l'aplicació amb la clau obtinguda en el primer pas.
4) Alinear el paquet APK.

Aquestes fases, si no es disposa d'un IDE, es fan per mitjà de diferents ordres, com *keytool*, *apksigner*, *zipalign*. En cas de tenir Android Studio, simplement haurem d'exportar l'aplicació per a ser distribuïda, i les accions de signatura i alineació són controlades per l'IDE.

Exemple de signatura i empaquetament d'una aplicació usant Android Studio



És important mantenir segures les claus privades amb les quals signem aplicacions públiques, ja que si algú malintencionat les aconsegueix pot canviar el codi i fer actualitzacions de la nostra aplicació sense el nostre permís, i tots els usuaris veurien que és una actualització més.

Si es publica una *app* d'Android Wear, hem de signar un paquet a part i incloure'l dins d'una *app* per a dispositius mòbils, ja que els dispositius *wearables* no poden instal·lar *apps* per si sols.

6.2. Versions de les aplicacions

Les aplicacions de l'Android permeten gestionar les versions de la mateixa aplicació. Això serveix per a mantenir informació del que conté cada aplicació, per a demanar nous permisos en cas de ser necessaris, o bé perquè altres aplicacions siguin conscients de la versió de què disposem actualment. Això últim és útil per a evitar incompatibilitats entre versions, de manera que una altra aplicació pot saber quina versió és compatible i preguntar quina és la versió actual d'aquesta aplicació o, generalment, servei.

La versió actual es defineix en el manifest i, en concret, amb dos atributs:

- **android:versionCode**. Corresponen a un enter que representa la versió del codi de l'aplicació i és relatiu a versions prèvies.

- **android:versionName**. És un text que identifica per als usuaris el nom de la versió. Pot ser quelcom, com succeeix amb l'Android, que identifiqui la versió de manera decimal: X.Y.Z, per a poder identificar clarament el nivell de canvis d'una versió respecte a una altra.

Es pot obtenir informació de la versió d'una aplicació concreta per mitjà de la classe *PackageManager* i el mètode *getPackageInfo(packageName, flags)*.

D'una versió a l'altra es poden voler actualitzar dades, com són les versions de l'SDK que estan suportades; igual que amb totes les aplicacions, ens permet definir els requisits mínims de l'aplicació. En cas d'actualitzar la versió i ser incompatible o requerir nous permisos, l'Android prendrà les accions corresponents per a garantir-ne el funcionament correcte segons les definicions del manifest.

Cal destacar que Android no fa servir aquesta informació de versió per a imposar restriccions sobre incompatibilitats amb tercers ni per a fer restriccions a versions anteriors o posteriors. Si es vol afegir aquesta funcionalitat, s'ha de programar dins l'*app* i controlar-ho i advertir-ho a l'usuari, si s'escau.

6.3. Consideracions prèvies a la publicació

Hi ha alguns punts que s'han de tenir en compte abans de passar a la publicació de l'aplicació per a usuaris finals:

- S'ha d'haver verificat l'aplicació amb dispositius reals. Com hem vist anteriorment, hi ha eines en l'SDK d'Android o bé fora d'aquest que ens ajuden a aquest propòsit. Això és important per a evitar la frustració dels usuaris i la possible mala reputació de la nostra aplicació a causa d'això.

- Afegir un contracte de llicència d'usuari final o *end user license agreement* (EULA) en cas de ser necessari, per a informar de les condicions de la nostra aplicació i en especial de les repercussions del contracte.

- En cas de tractar-se d'una aplicació no gratuïta, cal considerar afegir una llicència a l'aplicació, de manera que es pugui gestionar dins del sistema de llicències d'Android.

- Definir la icona i el nom de l'aplicació.

- Verificar que els nostres filtres del manifest són correctes, de manera que no es podrà instal·lar l'aplicació en un dispositiu o tipus de dispositiu que no hagi estat verificat.

- Eliminar del codi corresponent a depuració, sia de *logging* excessiu o altres fonts.

- Aconseguir les claus pròpies per a producció de biblioteques, com per exemple les claus de l'API del Google Maps.

- Fer la signatura per a distribuir l'aplicació.

- Provar l'aplicació signada en dispositius reals.

- Actualitzar els recursos de l'aplicació: assegurar que els recursos gràfics o multimèdia estan inclosos a l'*app* o bé disponibles en el servidor que els hostatja.

- Preparar el servidor i els serveis que oferirà a l'*app*, configurar-los adequadament i confirmar que funcionen com s'espera.

Google Play Licensing

L'Android aporta una biblioteca anomenada Google Play Licensing per a poder gestionar la propietat de les llicències per part dels usuaris. El procés de verificació es fa per mitjà de l'aplicació Google Play, que es comunica amb un servidor de llicències.

6.4. Publicació i seguiment

Quan ja tenim l'aplicació preparada per a ser distribuïda es pot transmetre de manera habitual (amb el corresponent fitxer *.apk*), o bé per mitjà de Google Play. Aquest canal de distribució afegeix molt valor, per la qual cosa és molt interessant conèixer-lo. Aquest canal ens ofereix poder arribar a milions de clients potencials amb una sola acció.

Per a publicar a Google Play es necessita un compte de desenvolupador de Google Play, que té un cost únic de vint-i-cinc dòlars. A partir d'aquí, si la nostra aplicació és de pagament, rebrem en el nostre compte de desenvolupador de Google Play el 70% dels guanys directes o indirectes de Google Play.

Algunes possibles opcions de publicació d'una aplicació a Google Play i els models de negoci que té són els següents:

Google Play Developer Console

És un espai on publicar l'*app* amb facturació integrada i gestionar els productes que ofereix per a adquirir dins l'*app*.

- **Una aplicació gratuïta**. Es poden baixar i utilitzar simplement sense cap restricció. En aquest cas se sol tractar d'aplicacions de suport a negocis principals fora de l'entorn mòbil, com poden ser bancs, o publicitat per comanda.

AdMob

AdMob és una companyia de publicitat mòbil comprada per Google el 2009. AdMob ofereix solucions d'anuncis per a plataformes natives com l'Android o l'iOS.

- **Una aplicació gratuïta amb publicitat**. Aquestes aplicacions solen ser molt populars, i el que aconsegueixen és que l'aplicació sigui totalment gratuïta, però a canvi l'usuari ha de veure certs anuncis, per exemple per mitjà d'AdMob.

Període de prova

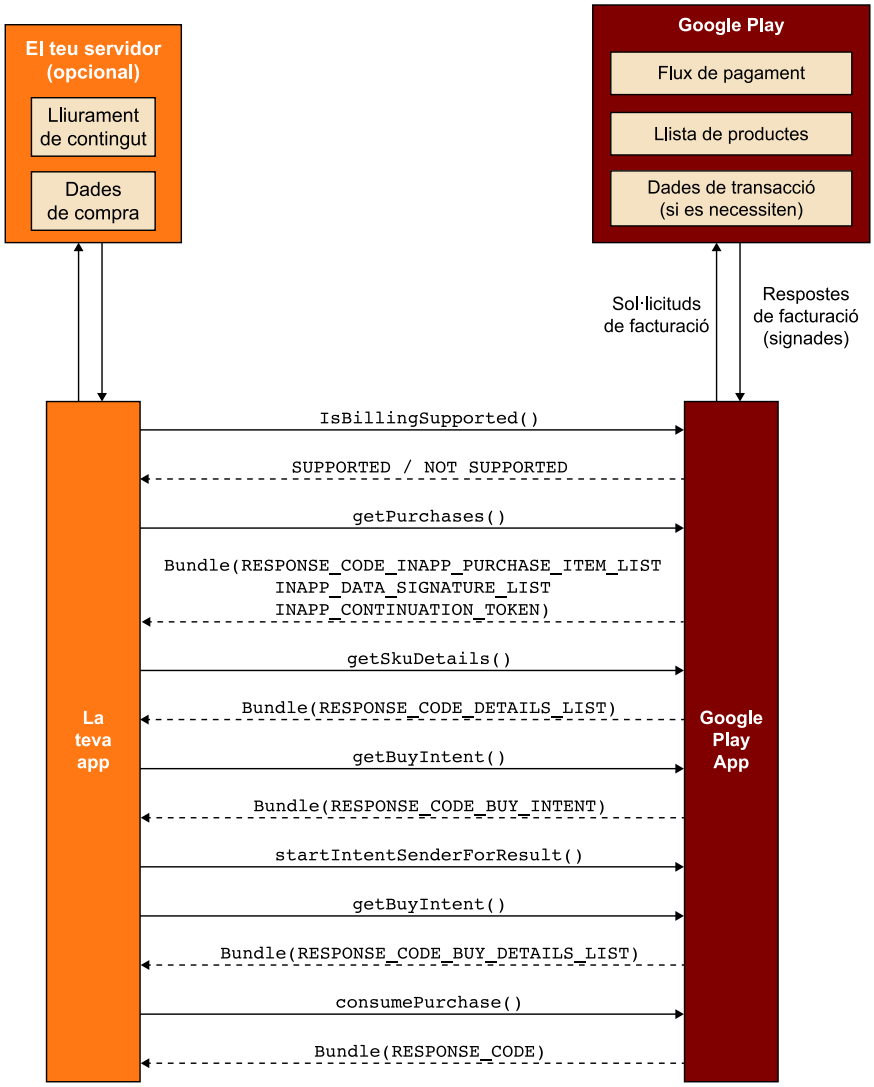
Els usuaris disposen de quaranta-vuit hores per demanar la cancel·lació d'una compra.

- **Una aplicació de pagament**. Són aplicacions que per a poder ser utilitzades requereixen que l'usuari les pagui. Generalment es permet a l'usuari final provar l'aplicació durant un curt període, amb la possibilitat de tornar-la i retirar el pagament.

- **Una aplicació en versió *lite***. Aquest tipus d'aplicacions són versions d'aplicacions de pagament que tenen unes funcionalitats reduïdes o eliminades. L'objectiu és atreure l'usuari final i que aquest compri la versió de pagament.

- **Pagaments per béns virtuals**. Això s'utilitza per a demanar pagaments en l'aplicació mateixa i gestionats per aquesta. L'Android actualment té una API per a això, anomenada *In-app billing*. Podem veure l'esquema de pagaments en la imatge a continuació.

Explicació de la compra de béns virtuals dins d'una aplicació per mitjà d'*In-app billing*



http://developer.android.com/

Totes les aplicacions que hem comentat es comercialitzen per mitjà d'un mercat. I a l'usuari es pot arribar de quatre maneres diferents:

- **Per mitjà de les llistes d'aplicacions més recents**. Aquestes estan organitzades per tipus d'aplicació (jocs o aplicacions) i categoria. En aquesta llista d'aplicacions entren totes les noves aplicacions, per la qual cosa el temps de permanència és limitat. Per això és important triar bé la categoria, sospesant el nombre d'aplicacions noves que arribaran.

- **Per mitjà de les cerques per paraules clau**. Un usuari en qualsevol moment pot escollir fer una cerca, i per tant hem de triar bé les paraules i potser incloure paraules internacionalitzades.

- **Per mitjà de les aplicacions *top***. Aquesta llista depèn de les baixades i de les valoracions dels usuaris. Aquesta llista és la més difícil de mantenir i alhora la que pot donar més beneficis; per a aconseguir que es mantingui

és important escoltar els nostres usuaris i fer millores suggerides per ells. També és bo fer actualitzacions habitualment.

Una aplicació en els mercats d'aplicacions sol tenir diversos components promocionals que decanten un usuari a utilitzar-la o no. Entre els més importants hi ha el nom i la descripció, les imatges o vídeo de la nostra aplicació, i finalment els comentaris i votacions dels usuaris.

Play Console

https://play.google.com/apps/publish/

Estadístiques de l'aplicació

Dins de Google Play, utilitzant la Play Console podrem seguir les estadístiques relacionades amb la nostra aplicació. Amb informació històrica d'instal·lacions de l'aplicació via Google Play i informació de la distribució dels dispositius, i podrem veure el percentatge d'instal·lacions que es fan amb cada versió de l'entorn de treball o fins i tot el dispositiu concret a què correspon. Això ens pot servir a l'hora de prioritzar les actualitzacions o millores de la nostra aplicació.

Resum

Aquest mòdul didàctic introdueix l'estudiant als principals conceptes del desenvolupament mòbil natiu basat en Android.

Introduïm el mòdul amb la història recent d'aquesta tecnologia i posem les bases sobre com s'estructuren les aplicacions.

Posteriorment hem vist els principals components comuns a totes o gran part de les aplicacions Android, com són les fases del cicle de vida, els *Intents*, el Manifest, etc.

A continuació hem aprofundit en la part relacionada amb la interfície gràfica, hem vist les possibilitats que ofereix Android per a combatre la fragmentació. I posteriorment hem mostrat a l'estudiant algunes de les parts més interessants del SDK com les aplicacions LBS, els magatzems de dades, o tecnologies com NFC o SIP.

Finalment hem vist les eines que hi ha per a facilitar-ne el desenvolupament, i també les eines necessàries per a la seva distribució i comercialització.

Activitats

1. Hem vist que les activitats o *activities* poden tornar al punt en el qual deixem l'activitat en cas de ser interrompuda, però n'hi ha algunes d'especials que no tenen aquest comportament. Quines són i per què?

2. Quan invoquem una nova activitat en la mateixa tasca, l'activitat sol estar associada a la tasca, i pertany plenament a la seva pila d'activitats, però hi ha alguns casos en els quals això no és del tot cert. Quins són aquests casos i què aporten?

Glossari

API *f* *Application programming interface.*

ART *f* Acrònim d'*Android Runtime*, la màquina virtual Java que utilitzen les versions d'Android més recents per a executar les aplicacions.

bytecode *m* Codi format per instruccions executables independents del maquinari final de la màquina.

Dalvik *f* Màquina virtual Java que s'utilitzava en les versions inicials d'Android per a executar les aplicacions. Actualment ha estat substituïda per ART.

depuració (*debug*) *m* Procés que identifica i corregeix errors de programació.

controlador (*driver*) *m* Component de programari responsable d'oferir accés a un component maquinari.

escoltador (*listener*) *m* Part del patró de disseny *Observer*, que és notificat quan succeeix un esdeveniment pel qual ha registrat interès.

esdeveniment *m* Acció que s'inicia generalment fora de l'àmbit d'aplicació d'un programa i que maneja un fragment de codi dins del programa.

GPS (*global positioning system*) *m* El sistema de posicionament global determina la posició d'un objecte, persona o vehicle a tot el món, i pot arribar a precisions de centímetres, encara que normalment parlem de metres. Va ser desenvolupat i instal·lat pel Departament de Defensa dels Estats Units. Per a poder donar aquest servei s'utilitzen desenes de satèl·lits que giren al voltant de la Terra en una òrbita superior als 20.000 km. Alternatives al GPS: GLONASS, de la Federació Russa, i Galileo, de la Unió Europea.

IDE (*Integrated Development Environment*) *m* Entorn integrat de desenvolupament, conjunt d'eines que donen suport a la implementació, desplegament, test i depuració d'aplicacions.

jUnit *m* Conjunt de biblioteques utilitzades en programació per a fer proves unitàries d'aplicacions Java.

keyword *f* Paraula clau que es pot servir per a identificar un objecte per mitjà d'altres conceptes, especialment útil per a cerques.

layout *m* Distribució visual dels objectes gràfics en l'aplicació.

LBS (*location based service*) *m* Servei basat en la posició geogràfica de l'usuari.

programari intermediari (*middleware*) *m* Capa de programari encarregada d'independitzar el desenvolupament dels detalls que estan per sota.

modularitzar *v tr* Dividir una aplicació en mòduls.

múltiples fils d'execució (*multithreading*) *m pl* Aplicat a la indústria mòbil vol dir poder mantenir diverses aplicacions executant-se alhora.

Open GL *f* Especificació estàndard que defineix una API multillenguatge i multiplataforma per a escriure aplicacions que produeixin gràfics 2D i 3D.

Open Handset Alliance *f* Consorci d'empreses format per Broadcom Corporation, Google, HTC, Intel, LG, Marvell Technology Group, Motorola, Nvidia, Qualcomm, Samsung Electronics, Sprint Nextel, T-Mobile i Texas Instruments.

renderització *f* Procés de generació d'una imatge des d'un model.

retrocrida *f* Crida a un mètode o funció en què un dels paràmetres serà una referència al mètode o funció.

SQL Lite *m* Motor de base de dades relacionals.

UI (*user interface* o interfície d'usuari) *f* Interfície gràfica o interfície de comunicació amb l'usuari.

URI (*universal resource identifier*) *m* Identificador únic dins de l'aplicació. Els URL són un tipus d'URI.

WebKit *m* Plataforma per a aplicacions que funciona com a base per als navegadors Safari, Google Chrome, Epiphany o Midori, entre altres. Està basada originalment en el motor de renderització KHTML del navegador web del projecte KDE, el Konqueror.

XML (*extensible markup language*) *m* Llenguatge de marques extensible. Metallenguatge extensible d'etiquetes desenvolupat pel World Wide Web Consortium (W3C). És una simplificació i adaptació de l'SGML i permet definir la gramàtica de llenguatges específics (de la mateixa manera que l'HTML és al seu torn un llenguatge definit per l'SGML). Per tant, l'XML no és realment un llenguatge en particular, sinó una manera de definir llenguatges per a diferents necessitats.

Bibliografia

Enllaços d'Internet

http://developer.android.com/

http://appinventor.mit.edu/explore/

https://www.genymotion.com/