Pràctica introductòria #2 - Activitats i d'altres components fonamentals

Projecte Integrat de Software (2024/25)
Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

Introducció

Les *activitats* són components d'Android que corresponen a les pantalles de l'aplicació. Defineixen el conjunt d'elements gràfics que presenten la informació a l'usuari i el maneig dels esdeveniments causats per la interacció de l'usuari amb aquests elements. A més de les activitats, existeixen altres components fonamentals de les aplicacions Android: *serveis*, per executar tasques sense la part gràfica d'una disposició (execució en segon pla); *receptors d'emissió*, per escoltar i enviar esdeveniments del/cap al sistema; i *proveïdors de contingut*, per a compartir dades amb aplicacions de tercers.

Aprofundint en les activitats, parlarem del seu *cicle de vida*, de *transicions entre activitats* (anar d'una pantalla un altre), de l'ús de la *pila de retrocés* per a tornar a activitats anteriors i de la *lligadura de vista* que simplifica el mapeig entre la part gràfica de la disposició i el maneig programàtic.

Requeriments

- PI1 - Primers passos amb Android Studio

Objectius

1. Introduir les **activitats** (Sec. 1).
2. Conèixer les **parts d'una activitat** (Sec. 1.2).
3. Entendre el **cicle de vida** de les activitats (Sec.1.3).
4. Aprendre a llançar components amb **intencions** i transicionar entre activitats (Sec.1.4).
5. Manegar la **pila de retrocés** (Sec.1.5).

6. Introduir la **lligadura de vista** (o *view binding*) (Sec.1.6).

7. Veure, molt breument, el propòsit d'**altres components fonamentals**: *serveis*, *receptors d'emissió* i *proveïdors de continguts* (Sec.2).

Recursos

Us proporcionem:

- `AuthenticationExample-v2.zip`: projecte base d'Android Studio descarregable del campus virtual, per fer el seguiment d'aquesta PI i resoldre els exercicis proposats.

Enllaços a altres recursos que us poden servir d'ajuda:

- **Documentació per desenvolupadors Android** (Enllaç). Concretament:
  - Guies (Enllaç)
  - API (Enllaç)
  - Exemples (Enllaç)

1 Activitats

En aquesta secció, veurem que les activitats són – de fet – punts d'entrada a l'aplicació (Sec. 1.1); que consten de dues parts ben distingides: la disposició i el control (Sec. 1.2); que tenen un cicle de vida (Sec. 1.3); que la creació d'activitats i la transició a d'altres activitats es fa mitjançant les *intencions* (Sec. 1.4); que es poden recuperar activitats anteriors gràcies a una pila de retrocés (Sec. 1.5; i, finalment, que la manera més convenient d'accedir a la disposició de manera programàtica és la lligadura de vistes – dit també *viewbinding* (Sec. 1.6).

Tot i centrar-nos fonamentalment en les activitats, parlarem també – tot i que més breument – dels altres components: serveis, receptors d'emissió i proveïdors de contingut (Sec. 2).

1.1 Les activitat són punts d'entrada a l'aplicació

Mentre que les aplicacions d'escriptori tenen un únic punt d'entrada (p. ex. el mètode `main()`), en les aplicacions Android és comú tenir-hi múltiples punts d'entrada; potencialment, tants com activitats tinguem. Si s'accedeix a l'aplicació des del llançador d'aplicacions del sistema operatiu, el punt d'entrada serà una activitat (o pantalla) principal. Ara bé, a Android també és comú que una aplicació "invoqui" una altra aplicació, demanant-li (comunicant-li la *intenció*) de fer alguna cosa concreta diferent de mostrar la pantalla principal. Penseu amb el típic botó que ofereixen moltes aplicacions de compartir multimèdia en

una aplicació de xarxa social. Quan s'obra l'aplicació de la xarxa social, no s'obra la pantalla principal, sinó una pantalla secundària on hi apareix el contingut i altres camps per editar (un missatge associat) i el botó per confirmar la compartició.

Concretament, la definició de quines seran les activitats que serviran de punts d'entrada a la nostra aplicació es fa en el fitxer `manifests/AndroidManifest.xml`.

Definir l'activitat principal. Veiem un exemple del manifest d'una aplicació amb dues activitats, `FooActivity` (punt d'entrada principal) i `BartActivity` (no punt d'entrada):

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/
    android"
  xmlns:tools="http://schemas.android.com/tools">

  <application
    ...

    <activity
      android:name=".FooActivity"
      android:exported="true">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.
            LAUNCHER" />
      </intent-filter>
    </activity>

    <activity
      android:name=".BarActivity"
      android:exported="false" />

    ...
  </application>
</manifest>
```

Fixeu-vos en que les diferències que fan que una activitat pugui ser principal són que (1) l'activitat sigui exportada (`android:exported="true"`), és a dir, sigui visible des de fora de l'aplicació i (2) que defineixi un filtre d'intenció concret.

Entrarem més tard en què són les intencions i per què és defineixen els filtres. De moment, quedeu-vos amb que les intencions són objectes de missatgeria entre activitats d'una mateixa aplicació, entre activitats de diferents aplicacions o entre el sistema operatiu i les aplicacions. I, encara més important, que són el mecanisme que utilitzarem per transicionar entre activitats de la nostra aplicació.

1.2 Les parts d'una activitat: disposició i lògica de control

Anem a veure com s'implementen les activitats. En el tutorial anterior, haureu creat una aplicació amb una activitat MainActivity. Aquesta era una activitat molt senzilla que mostra un text "Hello World!" per pantalla. Ara bé, per a crear aplicacions més complexes i, per tant, amb activitats amb altres elements gràfics que "facin coses", haurem de conèixer millor les dues parts de les que es componen les activitats: la *disposició* i la *lògica de control*.

1.2.1 Disposició (o *layout*)

Si inspeccionem el contingut del directori `res/layout` de qualsevol projecte, veurem que **cada activitat té una disposició associada** i que cadascuna és un fitxer xml. Per exemple, en el cas de MainActivity, el fitxer de disposició és `res/layout/activity_main.xml`.

Editor de disposicions d'Android Studio. Quan obrim un fitxer xml de disposició, veurem el codi. Ara bé, Android Studio té integrat un editor de disposicions, el qual ofereix tres vistes: de codi, de disseny i la partida de codi/disseny. La vista de codi ens mostra, òbviament, el contingut del fitxer xml – a la Fig. 2. La vista de disseny – tal com il·lustra la Fig. 1 – mostra un editor gràfic interactiu per a facilitar la tasca del programador alhora d'afegir elements (*widgets*), modificar atributs dels elements, crear relacions de dependència entre ells, etc. I la vista partida mostra les dues coses alhora. Podeu canviar d'una a l'altra amb els botons de la part superior dreta: "Code", "Split" i "Design". Els canvis fets en la vista de disseny es reflecteixen immediatament en la de codi i viceversa.

Afegir elements gràfics (widgets) i establir-los atributs. En la vista de disseny, disposeu de dues subfinestres essencials que són la *paleta* ("Palette") i l'*arbre de components* ("Component tree") plegades a l'esquerra dins de la mateixa finestra de l'editor. Veure la Fig. 3. La ***paleta*** us ofereix tots els *widgets* que podríeu voler afegir a la disposició, com ara botons, etiquetes de text, etc. Entre aquests també s'inclouen les *disposicions d'arrengament* que permeten agrupar i posicionar els elements de vista, però també definir relacions de pertinença de manera jeràrquica. Mentre que la majoria de widgets són de tipus `View`, els arrengaments són `ViewGroup`. Veure Fig. 4. A la pràctica, això vol dir que si un botó (`View`) és part d'un arrengament (`ViewGroup`), en cas d'amagar-se l'arrengament, el botó quedaria també amagat. L'***arbre de components*** és, precisament, on podeu visualitzar, revisar i canviar aquestes relacions de dependència.

A més de la *paleta* i l'*arbre de components*, tenim la subfinestra dels ***atributs*** (o "Attributes") a la dreta – tal com es mostra a la Fig. 5) –, que permet canviar els valors de certs paràmetres del *widget* que tingueu seleccionat. Per exemple, "Hello World!" és el valor de l'atribut `text` d'aquest widget de tipus

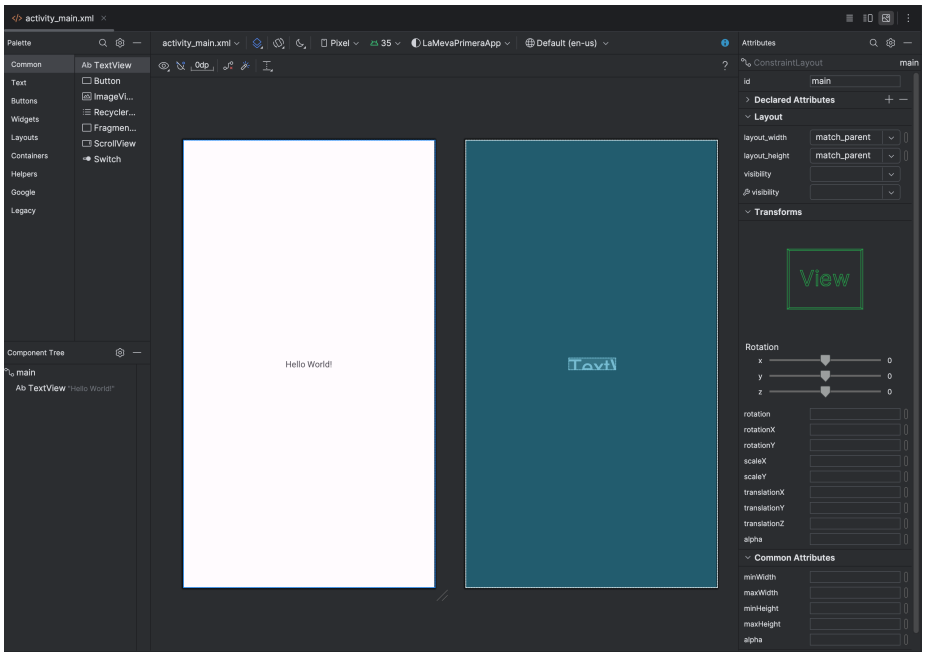

Figure 1: Vista de disseny de la disposició d'una activitat `res/layout/activity_main.xml`. La vista blava és el "blueprint", que mostra tots els widgets de la disposició independentment de la seva visibilitat inicial (si estan amagats quan comença l'activitat), mentre que la blanca només els que són visibles per defecte.

`TextView`[1]) que inclou la nostra disposició. Clicaríem primer a sobre del *widget* per a seleccionar-lo, canviaríem `text` i premeríem `<ENTER>` per aplicar-li el canvi.

Un dels atributs més importants dels widgets és l'identificador, `id`. Aquest atribut permet referenciar el widget des del codi Java de l'activitat. Per exemple, per modificar-ne el valors dels atributs en temps d'execució o per controlar un esdeveniment d'interacció (p. ex. el widget és un botó i l'usuari hi ha clicat).

Per afegir nous widgets a la disposició, només cal arrossegar-los des de la *paleta* fins a l'espai d'edició (o l'espai del blueprint), modificar-ne els atributs i assegurar-nos d'associar-lo a una disposició d'arrengament (com a mínim ha d'haver-n'hi una a cada activitat[2]).

Tipus de disposicions d'arrengament Aquestes disposicions – que no hem de confondre amb la disposició global de l'activitat (el fitxer .xml) – són objectes

[1] És interessant notar que `TextView` és una classe derivada de `View`. I que està contingut (o "penja") d'un *widget* de tipus `ConstraintLayout`, que és – aquest sí – de tipus `ViewGroup`.

[2] La disposició d'arrenjament per defecte de les activitats és un `ConstraintLayout`. Podríem canviar-la per una altra o afegir-n'hi més.

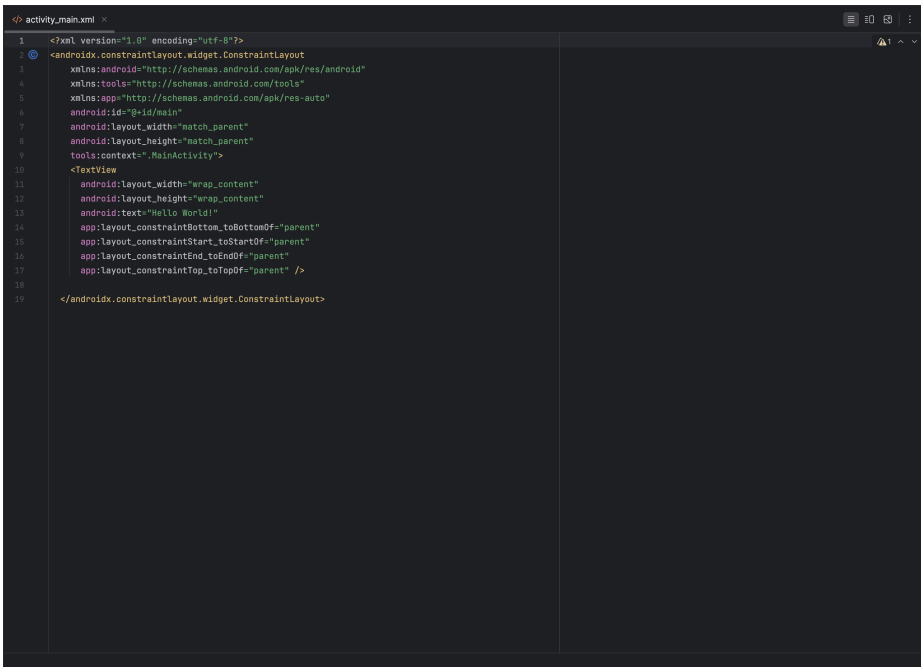

Figure 2: Vista de codi de la disposició d'una activitat `res/layout/activity_main.xml`.

que hereten de `ViewGroup` i posicionen els widgets (objectes`View`) de manera diferent en funció del tipus:

- `ConstraintLayout` posiciona els elements basats en restriccions respecte dels altres elements i els límits de la disposició (o *parent*). Els elements que formen part d'un `ConstraintLayout` han de tenir un mínim de restriccions definides respecte altres elements o els límits de la disposició. Substitueix el deprecat `RelativeLayout`.
- `LinearLayout` (horitzontal o vertical) posiciona els elements en una fila o columna on només cal indicar la mida relativa de cada element respecte la mida del parent.
- `FrameLayout` situa tots els elements superposats i centrats. Per a visualitzarlos alternats els amaguem i mostrem segons convinguin.
- `TableLayout` situa els elements en una taula amb múltiples files i columnes.

No hi ha restriccions a l'hora que una disposició d'arrengament en pugui contenir una altra. Per exemple, tenim el `ConstraintLayout` i, dins d'aquest, podem tenir-hi un `LinearLayout` (vertical). Les podem combinar jeràrquicament per obtenir l'arrengament desitjat.

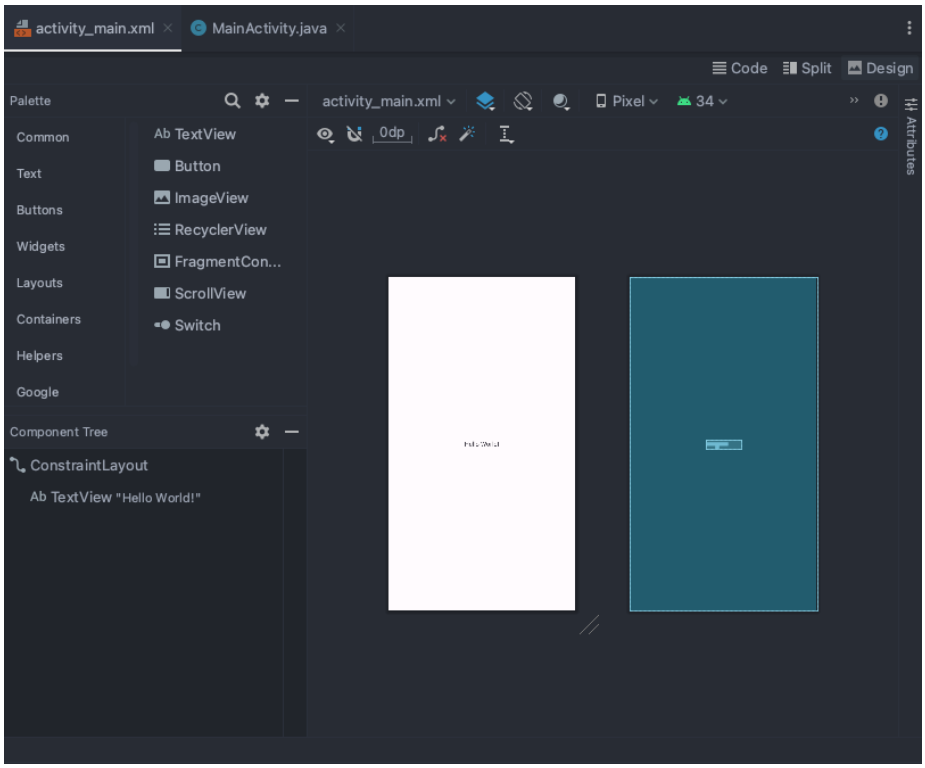

Figure 3: Subfinestres de la vista de disseny de la disposició: paleta i arbre de components. (Si apareixen "plegades" a la dreta, fer-hi clic per a desplegar-les.)

Finalment, remarcar que aquestes disposicions serveixen per a organitzar un número relativament petit i fixat (en temps d'execució) d'elements. Per exemple, disposar botons, camps d'entrada de text, etc. Si volguéssim mostrar una llista o taula d'elements amb un número gran i dinàmic d'elements, faríem servir un component anomenat `RecyclerView`. Això ho explicarem en una de les següents pràctiques introductòries.

Arribats aquest punt, podeu realitzar l'**Exercici 1** de la Sec. 2.3.

1.2.2 Lògica de control i caça esdeveniments d'interacció

La lògica de control de les activitats s'implementa en els fitxers Java. En el cas de la disposició `res/layout/activity_main.xml`, aquest seria el fitxer `MainActivity.java` (on hi definim la classe `MainActivity`). Aquí és on podem modificar la informació que mostren els elements gràfics (els widgets) en temps d'execució i també caçar-hi els esdeveniments causats per la interacció de l'usuari amb determinats elements.

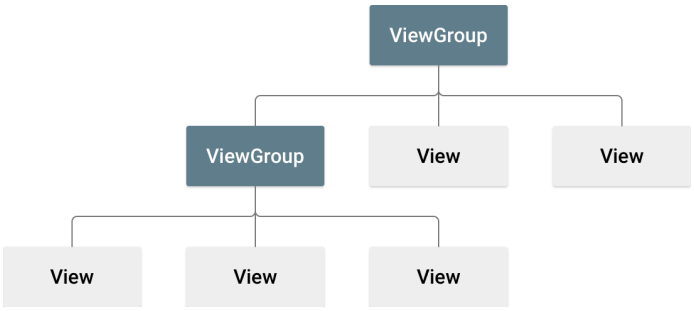

Figure 4: Jerarquia entre els elements de tipus `View` i de tipus `ViewGroup`. Font.

Classes Java de les activitats. Tota classe que representi una activitat ha d'heretar de la classe `AppCompatActivity`, i ha de sobrecarregar-ne almenys el mètode amb signatura `void onCreate(Bundle savedInstanceState)`. Aquest és el primer mètode que es crida (automàticament) quan una activitat inicia el seu cicle de vida. En la Sec.1.3 veurem el cicle de vida i quins mètodes s'invoquen quan l'activitat passa per les diverses etapes del cicle. Per ara, centrem-nos exclusivament en què fer posteriorment a l'inici de l'activitat, que és quan es crida l'`onCreate`.

En el mètode `onCreate` és on (1) fem accessibles els *widgets* de la disposició inicialitzant objectes Java que els representin i (2) definim, implementem i associem escoltadors pels esdeveniments dels *widgets*.

Accedir als *widgets*. hem d'associar la part de disposició i la lògica:

```
1    setContentView(R.layout.activity_main);
```

Estem indicant al codi Java on ha de trobar la seva disposició amb `R.layout.activity_main`. La classe `R` és una classe estàtica generada en temps de compilació, que conté constants que fan referència als recursos de l'aplicació. En aquest cas, `R.layout.activity_main` apunta al fitxer de disseny `activity_main.xml`, ubicat a `res/layout/`.

Ara ja podem rescatar qualsevol *widget* de la disposició (p. ex. un `Button`), fent:

```
1    Button btn = findViewById(R.id.idDelBoto);
```

on `idDelBoto` és l'identificador del botó assignat mitjançant l'atribut `id` en la disposició.

Caça d'esdeveniments de *widgets*. Podrem caçar l'esdeveniment de quan es premi aquest botó i definir-li un comportament associat, fent:

```
1    btn.setOnClickListener(new View.OnClickListener() {
```

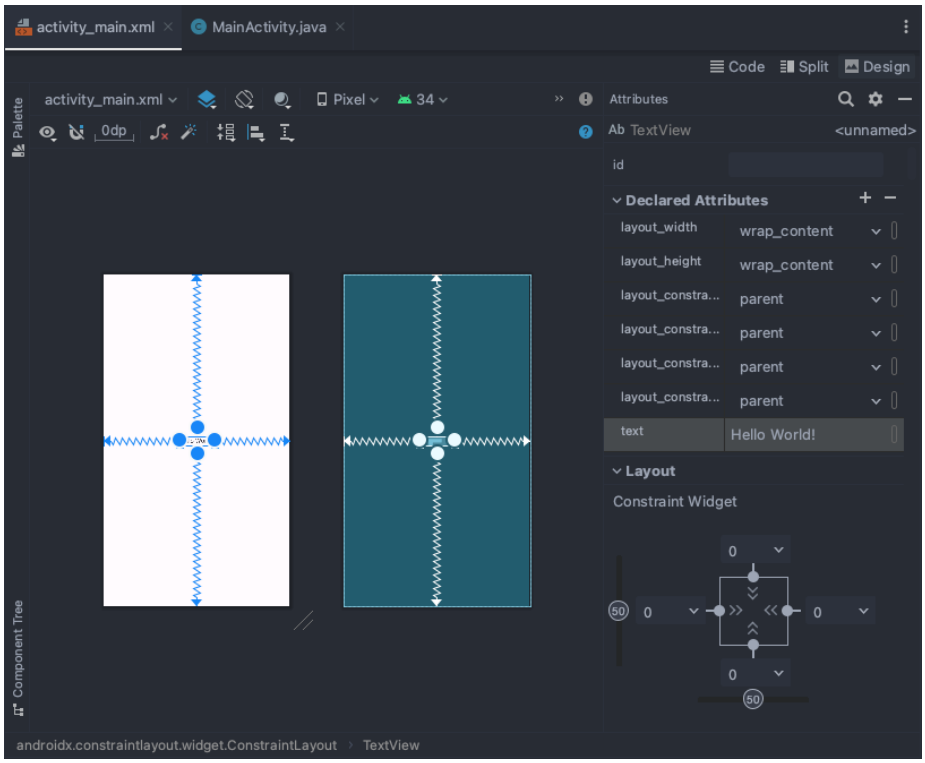

Figure 5: Subfinestra d'atributs que permet canviar els valors dels atributs d'un widget prèviament inclòs (i seleccionat) en la disposició de l'activitat.

```
    @Override
    public void onClick(View view) {
      // Implementació del comportament de delegació
    }
});
```

Aquí, en el pas de paràmetres del mètode `setOnClickListener` estan passant diverses coses alhora: definim una classe anònima que hereta de la classe abstracta `View.OnClickListener` (classe interna de `View`) a la qual li implementem el mètode abstracte `onClick` i creem una instància d'aquesta classe anònima per passar-li al mètode `setOnClickListener`. I el codi del mètode sobrecarregat `onClick` serà la lògica de tractament de l'esdeveniment que farà la nostra activitat - que és on hi escrivim aquest codi - quan `btn`, internament, cridi aquest mètode.

El mètode `onClick` té el paràmetre `view` que és – polimòrficament – el propi `btn`. Es fa el pas de paràmetre perquè podriem decidir implementar l'escoltador en una classe separada de l'activitat i no anònimament com hem fet aquí. En aquest supòsit, no tindriem accés a `btn`, perquè és una variable que

només té sentit en la classe de l'activitat.

Finalment, us podria interessar saber que com que la classe `View.OnClickListener` només conté un mètode abstracte (`onClick(...)`), podem simplificar el codi mitjançant l'ús de les funcions lambda de Java:

```
btn.setOnClickListener(view -> {
    // Implementació del comportament de delegació
});
```

Això és perquè el compilador ja sap quina classe espera el mètode `setOnClickListener`, que estem sobrecarregant l'únic mètode de la classe (`onClick`) i que el seu retorn `view` és de tipus `View`.

Un cop aquí, podeu fer l'**Exercici 2** de la Sec. 2.3.

1.3 El cicle de vida d'una activitat

Quan una activitat és llançada, ja sigui la principal o qualsevol altra, es criden una sèrie de mètodes d'aquesta (heretats de la classe *AppCompatActivity*). Fins ara, només hem vist l'`onCreate(...)`, però n'hi ha d'altres. Veure 6.

Explicació del cicle de vida. Quan una activitat és llançada, s'invoquen els mètodes `onCreate(...)`, `onStart(...)` i `onResume(...)` abans que aquesta – efectivament – estigui activa en pantalla i preparada, és a dir, visualitzi la informació i escoltant interaccions. Un cop sigui activa, continuarà així fins que una altra activitat sigui llançada i passi a primer pla (relegant l'activitat anterior a un segon pla) o l'aplicació acabi per ordre del sistema operatiu. Quan una activitat passa a segon pla, es crida el mètode `onPause(...)`. I si, a més, l'activitat del segon pla deixa de ser visible, es crida també `onStop(...)`. *Com pot una activitat estar en segon pla, però encara ser visible*? Doncs en el cas que la nova activitat sigui flotant i no ocupi tot l'espai de pantalla, deixant l'anterior activitat parcialment visible. Sigui com sigui, retornat a la primera activitat, es cridarà el mètode `onResume(...)` si aquesta havia sigut pausada o `onRestart(...)` seguit de `onStart(...)` i `onResume(...)` si havia sigut aturada. I quan una aplicació ha estat aturada, el sistema pot decidir destruir-la per falta de recursos (p.ex. memòria escassa en el dispositiu), causant la crida de l'`onDestroy(...)`. De ser així, quan es torni a llançar l'activitat, s'haurà d'iniciar un nou cicle de vida, tornant a passar per l'`onCreate(...)`.

Sobreescriptura dels mètodes del cicle de vida. La necessitat de sobreescriure alguns d'aquests mètodes quan implementem una activitat dependrà de les funcionalitats que aquesta hagi de brindar. Per exemple, en una aplicació de reproducció de vídeo per streaming, l'`onPause` s'encarregaria de pausar la reproducció del vídeo. Si a més, s'arriba a aturar l'activitat amb l'`onPause` podríem deixar de fer buffering de nous trossos del vídeo amb la previsió de que

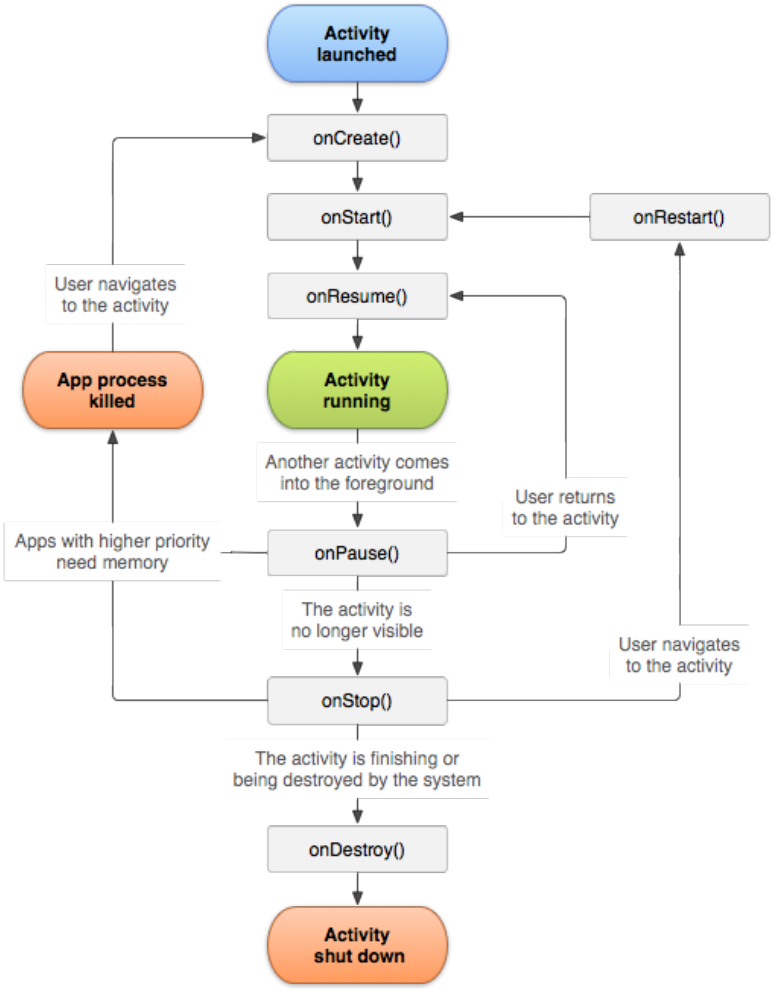

Figure 6: Cicle de vida de les activitats. Imatge de: `https://developer.android.com/guide/components/activities/activity-lifecycle`.

possiblement l'usuari no segueixi visualitzant el vídeo de manera immediata i estalviar-li dades mòbils.

Fins ara, hem treballat amb una sola activitat. I tot i que una activitat pugui oferir un nombre de funcionalitats, altres les oferiran altres activitats diferents. Evidentment no ens interessa tenir una activitat Déu i, per tant, buscarem repartir les funcionalitats entre les activitats que creiem convenients. Necessitem doncs una manera de canviar d'una activitat a una altra segons la funcionalitat que l'aplicació hagi d'oferir a l'usuari amb *intencions*.

1.4 Intencions

Les ***intencions*** són objectes de missatgeria que permeten demanar de realitzar accions als components fonamentals (entre els quals s'inclouen les activitats). Llançar des d'una activitat `A` una altra activitat `B`, llançar un nou servei `S` que realitzi una tasca per `A` o enllaçar-se a un servei ja existent o entregar un missatge a algun receptor d'emissions. Aquests són els exemples més rellevants de l'ús d'intencions, tot i haver-n'hi d'altres. Les intencions enviades cap a un component de la nostra aplicació del qual en coneixem un nom són les *intencions explícites*. Quan es tracta d'enviar una intenció a un component exposat per una altra aplicació, del qual no en coneixem el nom, podem fer ús de les *intencions implícites*.

1.4.1 Intencions explícites

S'utilitzen per a llançar activitats o serveis de la nostra mateixa aplicació.

Llançament d'activitat amb intenció explícita. Si volem anar d'una activitat `A` a una activitat `B`. Des del codi d'`A`, fem:

```
1    Intent intent = new Intent(
2      this,  // A.this si estem implementant un listener anònim
3      B.class
4    );
5    startActivity(intent);
6    finish(); // opcionalment
```

Això fa que s'iniciï la nova activitat `B` i `A` queda pausada i guardada en la *pila de retrocés*. A través d'aquesta pila, podem recuperar activitats anteriors quan es prem el típic botó de retrocés (◀ o *back*) o, en versions més modernes d'Android, un gest d'anar enrera. Ara bé, si volguéssim **impedir reprendre l'activitat** anterior `A` quan som a `B`, a més de `startActivity(intent)` hauríem d'invocar immediatament després el mètode `finish()`. Parlem amb més detall sobre la pila de retrocés a la Sec.1.5.

Llançament de serveis Android amb intenció explícita. En canvi, si volguéssim llançar un servei de descàrrega `S` des de l'activitat `A`:

```
    Intent intent = new Intent(
      this, // A.this en el context d'un listener
      S.class
    );
    intent.setData(Uri.parse(fileUrl));
    startService(intent); // enlloc de: startActivity(intent);
```

És poc probable que hagueu de fer ús dels serveis per el vostre projecte, però en cas que fos així tingueu en compte que poden existir implementacions de serveis de més alt nivell. Per exemple, per un servei de descàrregues, tindríeu el `DownloadService` (Enllaç) que valdria la pena mirar.

1.4.2 Intencions implícites

Les intencions implícites serveixen, normalment, per a demanar una acció a aplicacions de tercers o per oferir a aplicacions de tercers la realització d'una acció per part de la nostra aplicació.

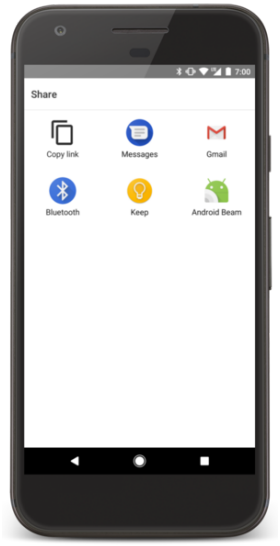

Figure 7: Selector d'aplicacions que mostra quines aplicacions instal·lades ofereixen la resolució d'una intenció amb determinada acció demanada.

Quan es tracti de **demanar realitzar una acció** a una aplicació tercera, serà la nostra qui llenci l'intent per iniciar la petició. Quan això passi, el sistema obrirà un selector on es mostraran totes les aplicacions amb un component que defineixi un `intent-filter` que especifiqui la capacitat de realitzar l'acció. Veure la Fig. 7.

Un exemple seria el d'enviar una imatge des de la nostra activitat `FooActivity` mitjançant una intenció amb l'acció `ACTION_SEND`, amb l'esperança

que un component d'una altra aplicació filtri intents amb una acció d'aquest tipus:

```
    Uri imageUri = ...

    Intent intent = new Intent();
    intent.setAction(Intent.ACTION_SEND);
    intent.putExtra(Intent.EXTRA_STREAM, uri);
    intent.setType("image/*");

    try {
        startActivity(Intent.createChooser(intent, "Share with
            ..."));
    } catch (ActivityNotFoundException e) {
        // En cas que cap app tingui una activitat que ofereixi
        // l'acció "ACTION_SEND"
    }
```

on `uri` seria la ruta al fitxer[3] de la imatge, `image/*` una restricció sobre el tipus de dades que enviarà `intent` i `*` el format de la imatge – en aquest cas, qualsevol format.

Si volguéssim **oferir una acció** tal com aquesta en la nostra aplicació, ho faríem mitjançant `<intent-filter>`.

```
<activity
    android:name=".BarActivity"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT
            " />
    </intent-filter>
</activity>
```

on amb l'etiqueta `<action>` definim un tipus d'acció i amb `<category>` simplement diem que filtrarem accions associades a intents implícits.

En cas de necessitar-ho, trobareu més informació sobre els intents a `https://developer.android.com/guide/components/intents-filters#java`.

1.5 Pila de retrocés (o *backstack*) d'activitats

A Android, cada cop que hi ha un canvi d'activitat, l'activitat anterior es guarda a la pila de retrocés. Si hem anat des d'una primera activitat `A` a `B`, a la pila de retrocés hi tindrem `A`. Permetent que, un cop a `B`, poguem tornar a `A` amb

[3] Des de certa versió d'Android, no es pot accedir directament als fitxers del dispositiu, sinó que s'ha d'utilitzar un proveïdor de contingut i establir permís en el manifest de l'aplicació. Veure `https://developer.android.com/reference/android/support/v4/content/FileProvider`.

el botó de retrocés o amb un gest (en les versions més noves d'Android). I, un cop haguéssim tornat a `B`, `A` seria destruïda.

Si volem impedir que l'usuari retrocedeixi a una activitat anterior, hem vist que podem destruir-la quan fem la transició `A -> B` amb `finish()`. *En quins casos ens pot interessar?* Imagineu que `A` és l'activitat principal d'una aplicació i que `B` és una activitat on l'usuari realitza i confirma una comanda. Un cop confirmada i acabada `B`, té sentit destruir-la i llançar de nou `A`. *Ara, sent a `A`, volem que l'usuari pugui tornat a l'activitat `B` i re-confirmar la mateixa comanda?* Sabem que per evitar-ho podem fer anar el `finish()` en el moment de llançar `B`.

Ara bé, imagineu que hem passat per diverses activitats per a realitzar una comanda, essent `A` una pantalla de logueig, `B` l'activitat principal de l'aplicació i `C-D-E` un seguit d'activitats involucrades en la realització de la comanda. Seguint amb la idea anterior, direm que quan acabem `E`, tornem a `B`. De manera més gràfica:

```
A -> B -> C -> D -> E -> B
```

Si féssim `finish()` en la transició `E -> B` i estant a `B`, un retrocés no ens portaria `E`, però sí a `D`. *Per què?* Doncs, perquè no havíem fet `finish()` en la transició de `D -> E`, sinó només en la `E -> B`. Podríem haver-lo fet en la `D -> E`, però això ens impediria – de manera innecessària – retrocedir entre les activitats de realització de comanda. De fet, és només quan fem `E -> B` que no volem que es pugui tornar a `E`, `D` o `C`. Aquí és quan cobra sentit la manipulació de la pila de retrocés per eliminar-ne diverses activitats de cop.

Per treure activitats anteriors fins un cert punt (fins que trobem de nou l'activitat a la que hem retornat) de la pila de retrocés, durant la transició de `E -> B` fem:

```
1  Intent intent = new Intent(E.this, B.class);
2  intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP | Intent.
       FLAG_ACTIVITY_SINGLE_TOP);
3  startActivity(intent);
4  finish();
```

Combinant els flags `FLAG_ACTIVITY_CLEAR_TOP` i `FLAG_ACTIVITY_SINGLE_TOP` amb `finish()`, aconseguim netejar la pila fins a trobar-nos la mateixa activitat a la que retornem (`B`) i, alhora, assegurar-nos que només hi hagi una instància d'`B`. Si no especifiquem `FLAG_ACTIVITY_SINGLE_TOP`, estarem creant una nova instància de `B` a més de la que ja hi havia a la pila (la que s'havia creat quan havíem passat pel flux inicial `A -> B -> C -> D -> E`).

Teniu més informació sobre la manipulació de la pila de retrocés a `https://developer.android.com/guide/components/activities/tasks-and-back-stack`.

1.6 Lligadura d'element vista (o *view binding*) en les activitats

Quan tenim molts elements en una disposició, als quals hem d'accedir des del codi Java de l'activitat, es fa feixuc haver de declarar variables i inicialitzar cadascun dels elements amb `findViewById`. Considereu el següent exemple d'activitat (ignorant el mètode `performLogInButtonAction`):

```
public class LogInActivity extends AppCompatActivity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_log_in);

    /* Obtenció elements de la vista */
    EditText etLoginUsername = findViewById(R.id.
        etLogInUsername);
    EditText etLoginPassword = findViewById(R.id.
        etLogInPassword);
    Button btnLogIn = findViewById(R.id.btnLogIn);
    // Altres findViewById ...

    /* Setejament d'escoltadors */
    btnLogIn.setOnClickListener(v -> {
      /* Obtenció valors dels camps emplenats per l'usuari */
      String username = etLoginUsername.getText().toString();
      String password = etLoginPassword.getText().toString();
      /* Crida un mètode auxiliar per a fer la crida al servei
          */
      performLogInButtonAction(username, password);
    });

    ...
  }
  ...
}
```

Aquí tenim 3 elements de la UI (`etLoginUsername`, `etLoginPassword` i `btnLogIn`), però podrien ser-ne bastants més. Aquestes declaracions i inicialitzacions de variables per a representar els *widgets* ens el podem estalviar amb *view binding*.

Primer haurem d'afegir al fitxer `Gradle Scripts > build.gradle (Module :app)`, les següents línies (tenint en compte que `android {...}` ja existeix):

```
android {
    ...

    buildFeatures {
```

```
        viewBinding true
    }
}
```

Això crearà una classe *binding* per a cada activitat que tingueu al projecte de manera totalment automàtica. Si la classe es diu `LogInActivity` (i el fitxer de disposició, activity_log_in.xml), la classe de lligadura serà `ActivityLogInBinding` i tindrà com atributs públics els elements de la disposició (ja inicialitzats).

Anem, doncs, a modificar l'activitat:

```
public class LogInActivity extends AppCompatActivity {

  /* Classe pel Data Bindinng */
  private ActivityLogInBinding binding;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    binding = ActivityLogInBinding
      .inflate(getLayoutInflater());
    setContentView(binding.getRoot());

    /* Setejament d'escoltadors */
    binding.btnLogIn.setOnClickListener(v -> {
      /* Obtenció valors dels camps emplenats per l'usuari */
      String username = binding.etLogInUsername.getText().
          toString();
      String password = binding.etLogInPassword.getText().
          toString();

      authenticationService.logIn(username, password);
    });

    ...
  }
  ...
}
```

Mireu com els elements de la UI s'accedeixen fàcilment fent `binding.idElement`, on `idElement` és l'atribut `android:id` assignat al fitxer de disposició.

Noteu també com ha canviat la línia del `setContentView`, que ja no pren – directament – per paràmetre el recurs `R.id.activity_log_in`, sinó l'arrel del *binding*. L'arrel és, precisament, el `ViewGroup` pare de tots els elements de la disposició; típicament, el `ConstraintLayout` que conté tots els altres elements. I el *binding* sap quina és aquesta arrel perquè el mètode anterior `inflate` ha fet l'associació de l'activitat amb el seu fitxer de disposició (seguint

la resolució de noms) i ha utilitzat l'inflador rebut per paràmetre per **inflar**-lo[4].

2 Altres components fonamentals

Tot i que descriurem molt breument aquests altres tipus de components fonamentals d'Android, l'explicació del seu funcionament podria ser molt més extensa. Podeu trobar-ne més informació aquí: `https://developer.android.com/guide/components/fundamentals`.

2.1 Serveis d'Android

Permeten córrer processos en segon pla i, per tant, sense interfície gràfica. S'utilitzen per a processos amb un temps de procés considerable o treballar en comunicació amb d'altres serveis.

Existeixen, bàsicament, tres tipus de servei:

- **Primer pla (o *foreground*):** realitzen operacions que són notificades a l'usuari, però continuen funcionant encara que l'usuari no estigui utilitzant l'aplicació. Tenen un cicle de vida independent del component (p. ex. una activitat) que els ha creat. És recomanable utilitzar `WorkManager`[5] en molts casos en lloc de serveis en primer pla.
- **Segon pla (o background):** realitzen operacions en segon pla sense donar cap informació a l'usuari a través de notificacions. Tenen un cicle de vida també independent del component. S'haurien d'evitar en dispositius amb API 26 o superior a causa de les actuals restriccions de privacitat/seguretat del sistema.
- **Vinculat (o bounded):** ofereixen una interfície client-servidor que permet als components interactuar amb el servei, enviar sol·licituds, rebre resultats i fer-ho fins i tot a través de processos amb comunicació entre processos (IPC). No tenen un cicle de vida propi i utilitzen el cicle de vida del component que els ha vinculat. Es destrueixen quan tots els components es desvinculen.

Podeu trobar més informació sobre els serveis a: `https://developer.android.com/develop/background-work/services`.

2.2 Receptors d'emissió

Permet a l'aplicació escoltar i rebre esdeveniments cap al i des del sistema. Un exemple d'esdeveniment rebut pot ser el de que l'aplicació se n'adoni de que la pantalla del dispositiu s'ha apagat. En l'altre sentit, tindríem l'exemple

[4] Parsejar l'xml i crear-ne tots els `View`/`ViewGroup`.

[5] `https://developer.android.com/develop/background-work/background-tasks/persistent/getting-started`

d'una alarma programada des de la nostra aplicació d'alarmes. Quan l'alarma es programa, s'avisa al receptor d'emissions de manera que, encara que tanquem l'aplicació, la notificació d'alarma seguirà arribant al sistema per a que la faci sonar a l'hora programada.

Podeu trobar més informació sobre els receptors d'emissió a: `https://developer.android.com/develop/background-work/background-tasks/broadcasts`.

2.3 Proveïdors de contingut

Un proveïdor de contingut manega el conjunt de dades compartit de l'aplicació o permet accedir a les dades d'altres aplicacions mitjançant el seu propi proveïdor de contingut. Els proveïdors de contingut permeten no només accedir a dades, sinó també modificar-les des d'aplicacions terceres – si el proveïdor ho permet. El conjunt de dades compartit no està pensat per ser utilitzat com a base de dades, sinó per a guardar informació concreta i rellevant per a altres aplicacions. Serveixen com a punt d'entrada per part del sistema (o components de tercers a través del sistema). Quan s'hi publica contingut es fa amb un nom que permet assignar a la dada una URI concreta per a que els altres components hi puguin accedir.

Un exemple és el de fer una foto per a la nostra aplicació. En aquest supòsit, no serà la nostra aplicació la que implementi la funció de càmera. Sinó que el que tindrem serà una activitat a la nostra aplicació que demanarà a l'activitat de prendre foto d'una aplicació de càmera que faci la foto, la guardi i en generi una URI a la que el proveïdor de contingut de la nostra aplicació pugui accedir per obtenir-ne el contingut i fer-lo accessible a la nostra aplicació.

Trobareu més informació sobre els proveïdors de continguts a: `https://developer.android.com/guide/topics/providers/content-provider-basics`.

Exercicis

Els següents exercicis estan plantejats de manera incremental. L'objectiu és acabar afegint la funcionalitat de sign-up, a més de la de log-in ja continguda en el codi base.

Exercici 1. Afegir la nova activitat per fer sign-up (registrar un nou usuari), seguint aquests passos:

1. Afegir un botó de sign-up a la disposició de l'activitat de log-in (`LogInActivity`), que és l'activitat des de la que accedirem a la nova activitat de sign-up. Per afegir el botó, podeu arrossegar un *widget* de tipus `Button` des de la paleta de *widget*s a l'editor de disposicions. Un cop afegit, heu d'assignar-li l'identificador `btnSignUp`.

2. Crear l'activitat `SignUpActivity`. Premeu el botó dret a sobre del paquet `edu.ub.pis2324.authenticationexample.presentation` i fer New > Activity > Empty Views Activity i poseu-li de nom `SignUpActivity`. Us crearà el Java i el fitxer xml de disposició associat. Assegureu-vos que s'afegeix a `manifests/AndroidManifest.xml` també.

3. Editeu la disposició associada a la nova activitat `res/layout/activity_sign_up.xml`. Podeu afegir-hi els elements des de zero o, si us resulta més fàcil, copiar i modificar el codi xml de la disposició de log-in (`res/layout/activity_log_in.xml`). Feu que el sign-up demani, com a mínim, un nom d'usuari, contrassenya i la confirmació de la contrassenya. Penseu que és bona pràctica que les disposicions no defineixin directament valors constants pels seus elements (p. ex. el text d'un `Button` o d'un `TextView`), sinó al fitxer `res/values/strings.xml`. Mireu com està fet per `res/layout/activity_log_in.xml`. Recordeu d'afegir-li un botó finalment per a confirmar el sign-up.

4. Afegiu la lligadura de vista (o *view binding*) a la `SignUpActivity`. Per fer-ho, afegiu un `ActivitySignUpBinding binding` a la classe Java de l'activitat i infleu el binding tal com s'ha explicat a la Sec.1.6. Un cop fet, comproveu que els elements de la disposició `res/layout/activity_sign_up.xml` són accessibles a partir de l'atribut `binding`, fent `binding.idElementDisposicio`.

Exercici 2. Afegiu escoltadors als botons. Recordeu que a l'Exercici 1, heu afegit dos botons: (1) el botó de la `LogInActivity` que permet anar a la `SignUpActivity` i (2) el botó de la `SignUpActivity` que permet confirmar el sign-up quan s'ha emplenat la informació. Inspeccioneu el codi del botó que ja existia a la `LogInActivity` per veure com s'afegeix un escoltador a un botó. El comportament dels botons hauria de ser el següent:

- El botó (1) hauria de crear un `Intent` per llançar l'activitat de sign-up.
- El botó (2), en canvi, haurà de fer dues coses. Primer, cridar el mètode que implementa la lògica del sign-up, `signUp(String username, String password, String passwordConfirmation)` de la classe `AuthenticationService` (està declarat, però sense implementar), i segon, tornar a l'activitat de log-in. Tingueu en compte que el mètode `signUp(...)` validarà l'entrada i si tot és correcte, afegireu el nou usuari a l'atribut `MockClientsHashMap clients`.

Exercici 3. Feu que els mètodes `logIn` i `signUp` de la classe `AuthenticationService` retornin `void` i feu que "retornin" la informació a través d'uns escoltadors que declarareu a la mateixa classe:

```
public class AuthenticationService {
  ...

  public interface OnLogInListener {
    void onLogInSuccess(Client client);
    void onLogInError(Throwable throwable);
  }

  public interface OnSignUpListener {
    void onSignUpSuccess();
    void onSignUpError(Throwable throwable);
  }

  ...
}
```

Els passos a seguir són:

1. Copieu i enganxeu les declaracions de les interfícies al codi d'`AuthenticationService`.
2. Adapteu els mètodes de `logIn` i `signUp` per a que rebin, a més dels paràmetres actuals, un escoltador del tipus corresponent (com a últim paràmetre).
3. Aneu a les activitats des d'on crideu, respectivament, `logIn` i `signUp` i modifiqueu les seves crides per, com a l'últim paràmetre, passar-los els escoltadors. Creeu i definiu els escoltadors anònimament en el moment de passar-los (igual que com fem amb els `setOnClickListener`).

El sentit de "retornar" informació amb escoltadors, i no pel `return` dels mètodes, és preparar-nos per l'escenari en que hi hagi serveis que no responguin immediatament (asincronia). D'aquesta manera, quan passem l'escoltador seguim amb l'execució de l'aplicació, evitant el bloqueig suposaria quedar-se esperant el `return` d'un servei extern asíncron que no controlem.