Pràctica introductòria #4 - Vistes reciclables

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

Introducció

Un cas d'ús molt típic en el desenvolupament d'aplicacions és el de mostrar llistes d'elements per pantalla. En el context d'Android, si el nombre d'elements és petit i constant (p. ex. un menú d'opcions), podríem crear una disposició d'arrengament (p. ex. `LinearLayout`) amb els *widgets* necessaris per mostrar cadascun dels elements (p. ex. un `ImageView` i un `TextView` per cada element). *Però què passa si necessitem mostrar una llista dinàmica i extensa d'elements? I si, a més, no la podem encabir en pantalla i, per tant, ha de ser desplaçable?* Penseu en una xarxa social amb un mur de publicacions pràcticament infinit. Té sentit crear i mantenir en memòria tots els elements d'UI necessaris per a totes i cadascuna de les publicacions, quan la majoria ni tan sols s'estan mostrant a la pantalla? No només no és necessari, sinó que seria un malbaratament de recursos!

Per resoldre-ho, Android ofereix les *vistes (de llista) reciclables* (o `RecyclerView`), que permeten crear llistes desplaçables i eficients reutilitzant el mínim nombre d'elements d'UI necessari per mostrar només el subconjunt de dades que calgui mostrar segons el que cap a pantalla i la posició de desplaçament en la llista. Quan l'usuari desplaça la llista, la vista reciclable desvincula les dades dels elements d'UI i els hi associa les dades que han d'aparèixer-hi segons la nova posició de desplaçament, evitant així la creació innecessària d'elements d'UI i reduint el consum de memòria.

Requeriments

- PI1 - Primers passos amb Android Studio
- PI2 - Activitats i d'altres components fonamentals
- PI3 - Model–Vista–Model de Vista

Objectius

1. Conèixer vistes reciclables (`RecyclerView`) per a mostrar un número dinàmic i gran d'elements.
2. Entendre com respectar el patró MVVM quan es tenen vistes reciclables.
3. Saber caçar esdeveniments d'interacció amb els elements d'una vista reciclable i delegar-ne la lògica a l'activitat / model de vista.

Recursos

- `RecyclerViewExample.zip`: projecte base d'Android Studio descarregable del campus virtual, per fer el seguiment d'aquest guió i resoldre els exercicis proposats. Està basat en el codi de la solució de la PI3 (`MVVMExampleSolution.zip`) per a fer-ho incremental.

1 Llistes dinàmiques amb Android

A Android, hi ha dues maneres de mostrar llistes amb un número dinàmic d'elements, les **vistes de llista** i les **vistes (de llista) reciclables**. Aquestes funcionalitats són proporcionades per *widgets* anomenats `ListView` i `RecylerView`. Ambdues permeten desplaçament lliscant, la qual cosa vol dir que si desplacem el dit (verticalment o horitzontal) per sobre d'aquests *widgets*, no desplaçarem la pantalla, sinó la posició dins de la llista i, per tant, el subconjunt de dades que mostraran.

La diferència entre el `RecyclerView` i `ListView` rau, principalment, en la seva eficiència. `RecyclerView` té la capacitat de "reciclar" els objectes de tipus `View` que utilitza per mostrar els elements de la llista, mentre que `ListView` no. `ListView` crea tants `View` com elements tingui la col·lecció de dades a partir de la qual s'ha de nodrir. Vegeu la Fig. 1. Només en el cas que sapiguem que el nombre d'elements de la col·lecció és indefinidament petit, pot ser interessant valdre'ns d'una `ListView` perquè, a més, resulta més fàcil d'implementar. Altrament, optarem per una `RecyclerView`.

En aquesta pràctica introductòria, ens centrarem completament en les `RecyclerView`. En cas de necessitar `ListView`, podeu consultar els següents recursos:

- API: ListView
- Developer guide: Layouts in views > Fill an adapter view with data

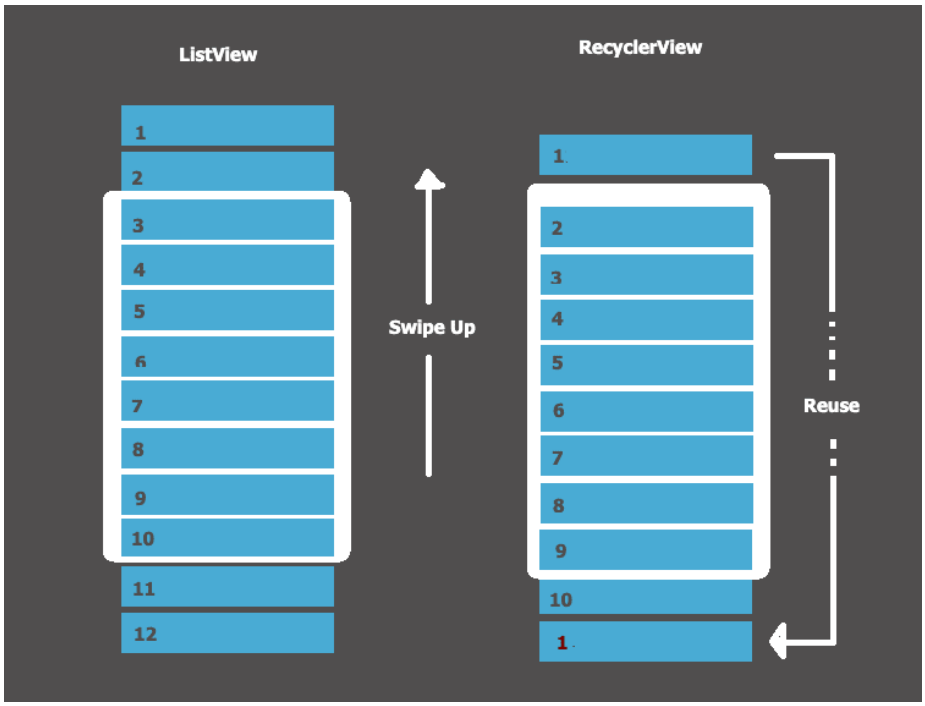


Figure 1: `ListView` vs `RecyclerView`. Amb les `RecyclerView` reaprofitem el conjunt estrictament necessari d'elements d'UI per a mostrar les dades, enlloc de tenir-ne un per cada dada. Imatge de: `https://dev.to/jbc7ag/recyclerview-or-listview-pros-cons-and-examples-with-kotlin-2nb2`

2 Vistes reciclables

La implementació d'una vista reciclable involucra diverses classes de l'*Android Sdk*:

- La **classe `RecyclerView`** és la implementació del *widget*. Igual que qualsevol altre *widget*, el podeu afegir a la disposició gràfica d'una activitat i accedir-hi programàticament a través del codi Java.
- Els elements de la vista reciclable són els *view holder*. Els *view holders* seran instàncies d'una classe concreta que haureu de derivar a partir de **classe abstracta `RecyclerView.ViewHolder`**. Aquesta classe abstracta té un atribut `View` que representa la disposició gràfica d'un element de la `RecyclerView` (Fig. 2a). I la vostra implementació concreta afegirà els atributs que representin els *widgets* del `View`, obtinguts amb `findViewbyId` – igual que fem a les activitats per accedir programàticament als elements de la disposició gràfica associada.

- La **classe abstracta `RecyclerView.Adapter<T>`** (on `T` serà la classe concreta dels *view holder*) que haureu d'estendre per implementar (1) la lògica de creació dels *view holder* i (2) el *binding* de les dades als atributs del *view holder* corresponent (segons la posició de desplaçament). Aquestes dues operacions es faran a petició de la `RecyclerView`, que és qui té el control de la posició de desplaçament i, per tant, determina quin *view holder* ha de mostrar quina dada.
- La **classe `RecyclerView.LayoutManager`**, o gestor de disposició d'arranjament de la vista reciclable, és una classe que permet definir la manera com es mostren els elements a la `RecyclerView`. Ofereix dues subclasses, `LinearLayoutManager` per una disposició lineal (horitzontal o vertical) dels elements o `GridLayoutManager` per disposar-los com una graella.

2.1 Exemple: una vista reciclable de productes

En una activitat `ShoppingActivity`, volem mostrar-hi els productes del catàleg d'una botiga en una `RecyclerView`. Veure Fig. 2b.

Els passos que hem de seguir i que il·lustrarem amb aquest exemple són: (1) crear una disposició gràfica pels elements de la vista reciclable, és a dir, els *view holders* (Sec. 2.1.1), (2) derivar un classe adaptador i de *view holder* concretes (Sec. 2.1.2), (3) afegir la vista reciclable a la disposició gràfica de l'activitat i accedir-hi programàticament per setejar-li l'adaptador i el gestor de disposició d'arranjament (Sec. 2.1.3).

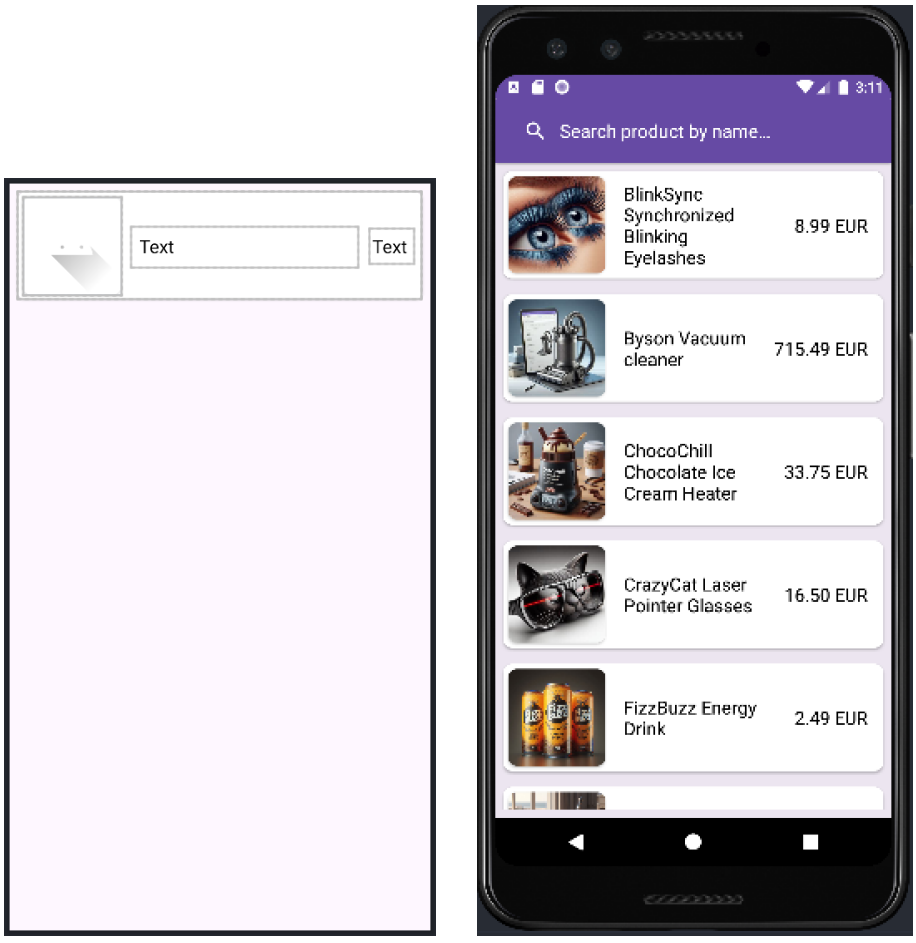
A més a més, veurem també el paper que juga el model de vista en el context de les vistes reciclables (Sec. 2.1.4) i com afegim escoltadors pels elements de la vista reciclable (Sec. 2.1.5).

En el codi base, tot això ja està implementat, però ho haureu d'entendre per fer els exercicis.

2.1.1 Creació de disposició gràfica pels elements de la vista reciclable (`item_rv_product.xml`)

S'ha de crear una disposició gràfica (xml) per representar els elements de la vista reciclable. Per a crear-la, s'ha de fer clic amb el botó dret sobre el directori `res > layout`, triar `New > Layout resource file` i posar-li un `File name` sense extensió (p. ex. `item_rv_product`). Un cop creada, es pot editar amb la vista de disseny de l'Android Studio per obtenir alguna cosa semblant a la Fig. 2a.

Podeu examinar `item_rv_product.xml` del codi base si ho creieu convenient. No l'incloem aquí per no allargar innecessàriament el guió. Només cal que



(a) `View` que mostrarà la informació d'un element de la `RecylerView` (vist en la vista de disseny de l'editor de disposicions d'Android Studio).

(b) `ShoppingActivity` amb un `RecyclerView` que mostra una llista de productes.

Figure 2: Exemple de `View` que mostra la informació dels elements i `RecyclerView` que els mostra.

sapigueu que la disposició que hem creat té els següents *widgets*: un `ImageView` (amb identificador `"ivProductImage"`), un `TextView` pel nom del producte (`"tvProductName"`) i un `TextView` pel preu (`"tvProductPrice"`).

2.1.2 Implementació de l'adaptador (`ProductRecyclerViewAdapter`)

Com que voldrem mostrar una llista de `Product`, hem de crear una classe adaptador que derivi de la classe abstracta `RecyclerView.Adapter<ProductR`

ecyclerViewAdapter.ProductViewHolder>, que anomenarem ProductRecyclerViewAdapter. Seguidament, hi hem de definir el constructor, un setejador per passar-li des de fora (des de l'activitat) les dades que volem que mostri, sobreescriure tres mètodes (@Override) de la classe abstracta i definir-hi una classe interna pels *view holder*.

Veiem, primer, el constructor de ProductRecyclerViewAdapter i el mètode setProductsData de **ProductRecyclerViewAdapter**:

```
1  public class ProductRecyclerViewAdapter extends RecyclerView.
       Adapter<ProductRecyclerViewAdapter.ProductViewHolder> {
2
3    private List<Product> productList;
4
5    public ProductRecyclerViewAdapter() {
6      super();
7    }
8
9    public void setProductsData(List<Product> productList) {
10     this.productList = productList;
11     notifyDataSetChanged();
12   }
13
14   ...
15 }
```

El més interessant és la invocació del mètode notifyDataSetChanged() (línia 11), que és el que s'encarrega d'avisar a la classe abstracta de que les dades han canviat i de que cal tornar a "pintar". Però abans de que es pugui "pintar" res, necessitarem sobreescriure onCreateViewHolder, onBindViewHolder i getItemCount i la implementació de la classe interna, ProductViewHolder, que implementa la classe abstracta RecyclerView.ViewHolder.

❗ L'adaptador no ha de modificar les dades, productList, sinó només tenir-ne la referència. Si algú les ha de modificar és l'expert en informació, és a dir, el model de vista (o l'activitat si no tinguéssim model de vista) per, així, mantenir la integritat i coherència de les dades.

Examinem la resta de la implementació de l'adaptador:

```
1  public class ProductRecyclerViewAdapter extends RecyclerView.
       Adapter<ProductRecyclerViewAdapter.ProductViewHolder> {
2    ...
3
4    @NonNull
5    @Override
6    public ProductViewHolder onCreateViewHolder(@NonNull
         ViewGroup parent, int viewType) {
7      View view = LayoutInflater
```

```
        .from(parent.getContext())
        .inflate(R.layout.item_rv_product, parent, false);

    return new ProductViewHolder(view);
  }

  @Override
  @Override
  public void onBindViewHolder(@NonNull ProductViewHolder
      holder, int position) {
    Product p = productList.get(position);

    String name = p.getName();
    String price = p.getPrice().toString();
    String imageUrl = p.getImageUrl();

    holder.tvProductName.setText(name);
    holder.tvProductPrice.setText(price);
    Picasso.get().load(imageUrl).into(holder.ivProductImage);
  }

  @Override
  public int getItemCount() {
    return (productList == null) ? 0 : productList.size();
  }

  public static class ProductViewHolder
      extends RecyclerView.ViewHolder
  {
    private final ImageView ivProductImage;
    private final TextView tvProductName;
    private final TextView tvProductPrice;

    public ProductViewHolder(@NonNull View itemView) {
      super(itemView);

      ivProductImage = itemView
        .findViewById(R.id.ivProductImage);
      tvProductName = itemView
        findViewById(R.id.tvProductName);
      tvProductPrice = itemView
        .findViewById(R.id.tvProductPrice);
    }
  }
}
```

Comencem per `ProductViewHolder`. Fixeu-vos com pel constructor s'injecta una `View`, que és la que havíem creat a partir de la disposició de la Sec. 2.1.1. I, des del mateix constructor, accedim als *widgets* de la `View`

via `findViewById` i els assignem a un atribut privat. Com veieu la implementació resulta senzilla. I no calen *getters*/*setters*, perquè els atributs privats d'una classe interna són accessibles des de la classe externa, que és qui hi haurà d'accedir.

Ara, anem a veure la sobrecàrrega dels mètodes:

- `onCreateViewHolder` crea la `View` a partir del fitxer de disposició (`item_rv_product.xml`) amb un `LayoutInflater`[1] i crea el *view holder* `ProductViewHolder`, injectant-hi la `View` pel constructor. Aquest mètode és cridat per la vista reciclable només al principi de mostrar la llista, tantes vegades com elements pot mostrar la vista reciclable en pantalla. I, a partir d'aquí, es reciclen.
- `onBindViewHolder` s'invoca sempre que un desplaçament provoqui que un element surti de pantalla i n'hagi d'entrar un altre en el seu lloc. Aquest mètode rep el `ProductViewHolder` que s'ha alliberat després del desplaçament de la llista per tal d'associar-li (fer-li *binding*) de la informació (els atributs) de la nova dada que cal mostrar. En aquest cas, un nou `Product`.
- `getItemCount` serveix per a que la vista reciclable i el seu `RecyclerView.LayoutManager` sàpiguen quantes dades té el conjunt de dades, `List<Product>`, setejat prèviament amb el mètode `void setProductsData(List<Product> productList)`.

Amb l'adaptador i els *view holders* preparats, anem a veure com s'inclou una `RecyclerView` a una activitat.

2.1.3 `RecyclerView` a l'activitat `ShoppingActivity`

Primer, caldrà incloure un widget `RecyclerView` a la **disposició de la `ShoppingActivity`**. Podem fer-ho a través del codi xml de la disposició gràfica (`res/layout/activity_shopping.xml`) o amb la vista de disseny de l'editor d'Android Studio. Si ho fem a través de l'xml, hi afegirem un codi com aquest:

```
1  ...
2
3  <androidx.recyclerview.widget.RecyclerView
4      android:id="@+id/rvProducts"
5      ...
6      app:layout_constrainedHeight="true" />
7
8  ...
```

[1] El `LayoutInflater` és capaç de crear un objecte `View` a partir d'un fitxer de disposició gràfica (.xml) en temps d'execució.

Alternativament, podeu fer servir l'editor de disposicions d'Android Studio i a la pestanya *Palette* de l'editor, buscar el widget a `Common > RecyclerView` i arrosegar-lo. En qualsevol cas, cal assegurar-se de tenir l'atribut `app:layout_constrainedHeight = "true"` a l'xml, ja que, sinó la `RecyclerView` pot ignorar les restriccions de la disposició de l'activitat que la contingui i ocupar-ne tot l'espai vertical.

Examinem, finalment, el codi de **`ShoppingActivity.java`**:

```
1  public class ShoppingActivity extends AppCompatActivity {
2
3    /* Custom adapter for the recycler view of products */
4    private ProductRecyclerViewAdapter rvProductsAdapter;
5    /* LayoutManager for the recycler view of products */
6    private RecyclerView.LayoutManager rvLayoutManager;
7
8    ...
9
10   @Override
11   protected void onCreate(Bundle savedInstanceState) {
12     ...
13     initRecyclerView();
14   }
15
16   private void initRecyclerView() {
17     rvLayoutManager = new LinearLayoutManager(this);
18     binding.rvProducts.setLayoutManager(rvLayoutManager);
19
20     rvProductsAdapter = new ProductRecyclerViewAdapter();
21     binding.rvProducts.setAdapter(rvProductsAdapter);
22   }
23
24   ...
25 }
```

Obtenim la referència de la vista reciclable, `binding.rvProducts`, per assignar-li, primer, el gestor de disposicions (línies 17-18); concretament, un gestor de disposicions lineal `LinearLayoutManager` (que, inicialitzat sense paràmetres, serà linealment vertical). I, en segon lloc, la instància de l'adaptador, `ProductRecyclerViewAdapter` (línies 20-21).

2.1.4 Vistes reciclables + MVVM

Quan tinguem un model de vista, aquest serà l'expert en informació de les dades. És a dir, les dades que necessita l'adaptador de la vista reciclable, provindran del model de vista. En el nostre exemple, aquestes eren un `List<Product>`. I l'intermediari a través del qual fem arribar-li les dades a l'adaptador és l'activitat. D'aquesta manera no trenquem el patró MVVM i mantenim més desacoblats

els diversos components.

A continuació, **ShoppingViewModel** amb la llista `List<Product>` i l'observable `productsState`:

```
1  public class ShoppingViewModel extends ViewModel {
2
3    private final ProductStoreService productStoreService;
4    private final List<Product> products;
5    private final MutableLiveData<List<Product>> productsState;
6
7    public ShoppingViewModel() {
8      super();
9      productStoreService = new ProductStoreService();
10     products = new ArrayList<>(); // no data initially
11     productsState = new MutableLiveData<>();
12   }
13
14   public LiveData<List<Product>> getProductsState() {
15     return productsState;
16   }
17
18   public void fetchProductsCatalog() {
19     /* For simplicity, assume call to productStoreService is
           synchronous */
20     List<Product> gottenProducts = productStoreService.getAll()
           ;
21
22     products.clear();
23     products.addAll(gottenProducts);
24     productsState.postValue(Collections.unmodifiableList(
           products));
25   }
26
27   ...
28 }
```

Tingueu en compte, que quan emetem un objecte a través d'un observable `LiveData`/`MutableLiveData`, no se'n fa còpia, sinó que passem la referència. Per tant, si passem una col·lecció de Java, estaria bé assegurar-nos de que no la modificarà cap altra classe. Una possibilitat és fer ús de `Collections.unmodifiableList(...)` (línia 24).

Tornem a la **ShoppingActivity**, per veure-hi el codi de l'observador que observa canvis en l'observable de la llista de productes del model de vista (línies 29-34) i, també, com l'envia a l'adaptador de la vista reciclable a través del mètode `void setProductsData(List<Product>)` que hi hem implementat anteriorment (línies 32):

```
1  public class ShoppingActivity extends AppCompatActivity {
2
3    ShoppingViewModel shoppingViewModel;
4    private ProductRecyclerViewAdapter rvProductsAdapter;
5    ...
6
7    @Override
8    protected void onCreate(Bundle savedInstanceState) {
9      ...
10     initRecyclerView();
11     initViewModel();
12   }
13
14   @Override
15   protected void onResume() {
16     super.onResume();
17     shoppingViewModel.fetchProductsCatalog();
18   }
19
20   ...
21
22   private void initViewModel() {
23     shoppingViewModel = new ViewModelProvider(this).get(
           ShoppingViewModel.class);
24
25     initObservers();
26   }
27
28   private void initObservers() {
29     shoppingViewModel.getProductsState().observe(
30         this,
31         products -> {
32           rvProductsAdapter.setProductsData(products);
33         }
34     );
35   }
36
37   ...
38 }
```

Apuntar que és important no oblidar-se d'invocar el `fetchProductsCatalog()` del model de vista, posteriorment a la seva inicialització i a la dels observadors, en algun punt del codi de l'activitat. Aquí hem triat fer-ho en l'`onResume` de l'activitat (línia 17). Sinó les dades no es carregaran i, per tant, la vista reciclable serà buida quan s'iniciï l'activitat.

2.1.5 Escoltar esdeveniments d'interacció amb els elements de les vistes reciclables

Si l'usuari interactua amb un element de la vista reciclable, està interactuant amb el `View` (i/o algun dels seus widgets) atribuït a un *view holder*, no pas amb l'adaptador, la vista reciclable o l'activitat que conté la vista reciclable. Per tant, la definició d'escoltadors d'esdeveniments d'interacció amb la `View` i/o els seus elements, també la farem als *view holder*.

Ara bé, la lògica de la interacció la podriem haver d'acabar delegant a l'activitat (o el model de vista). Malauradament, els esdeveniments d'interacció són **asíncrons** i, per tant, la comunicació des del *view holder* (on té lloc l'esdeveniment "primigeni" de clic) fins a l'activitat o model de vista, passant per l'adaptador, la haurem de fer a través d'un o més escoltadors i crides niuades dels seus *callback*.Un exemple de delegació en l'activitat seria un cas d'ús en què l'usuari cliqui un dels elements de la vista reciclable, els quals representen productes, per veure'n els detalls en una activitat dedicada. I un exemple de delegació en el model de vista podria ser un cas d'ús que impliqui eliminar productes de la llista; ja que, el model de vista n'és l'expert. En aquesta secció, veurem la implementació del primer cas d'ús i la del segon la deixem pels exercicis.

Exemple: veure els detalls d'un producte seleccionat

En primer lloc, setejem al `View` del *view holder* un primer escoltador, de clic (`View.OnClickListener`), implementat anònimament (línies 16-19), que en la implementació del seu *callback*: (1) obté la posició del *view holder* segons l'adaptador i (2) invoca el *callback* d'un segon escoltador `ProductViewHolder.OnPositionClickListener` al qual li passarem la posició obtinguda com a paràmetre. Veiem el codi del **`ProductViewHolder`** amb tot això que hem comentat:

```
1  public static class ProductViewHolder extends RecyclerView.
       ViewHolder {
2    ...
3
4    public interface OnPositionClickListener {
5      void onPositionClick(int position);
6    }
7
8    ...
9
10   public ProductViewHolder(
11       @NonNull View itemView,
12       OnPositionClickListener onPositionClickListener
13   ) {
14     ...
15
```

```
16        itemView.setOnClickListener(v -> {
17          int pos = getAdapterPosition();
18          onPositionClickListener.onPositionClick(pos);
19        });
20      }
21  }
```

El *callback* del segon escoltador l'haurem implementat en el moment de crear el `ProductViewHolder` i li haurem injectat pel constructor. Veiem la seva a la classe **`ProductRecyclerViewAdapter`** (línies 33-40):

```
1   public class ProductRecyclerViewAdapter extends RecyclerView.
        Adapter<ProductsRVAdapter.ProductViewHolder>
2   {
3     /* Referència a la llista que ha d'adaptar */
4     private List<Product> productList;
5     /* Escoltador de clics als productes */
6     private OnProductClickListener onProductClickListener;
7
8     /* Interficie per retornar Product clicat a l'activitat */
9     public interface OnProductClickListener {
10      void onProductClick(Product product);
11    }
12
13    /* Constructor modificat amb escoltador */
14    public ProductsRVAdapter(
15        OnProductClickListener onProductClickListener
16    ) {
17      super();
18      this.onProductClickListener = onProductClickListener;
19    }
20
21    @NonNull
22    @Override
23    public ProductViewHolder onCreateViewHolder(
24        @NonNull ViewGroup parent,
25        int viewType
26    ) {
27      View view = LayoutInflater.from(parent.getContext())
28          .inflate(R.layout.item_rv_product, parent, false);
29
30      return new ProductViewHolder(
31          view,
32          /* Implementació i injecció de l'escoltador propi al
                view holder */
33          new ProductViewHolder.OnPositionClickListener() {
34            @Override
35            public void onPositionClick(int position) {
36              /* Utilitzem la position per recuperar el Product
                    */
```

```
37              Product product = productsList.get(position);
38              onProductClickListener.onProductClick(product);
39            }
40          }
41      );
42    }
43
44    ...
45  }
```

Fixeu-vos, ara, que el *callback* d'aquest segon escoltador fa el següent: (1) recuperar el `Product` indexant la `List<Product>` amb la `position` rebuda (línia 37) i (2) passar el `Product` pel *callback* d'un tercer escoltador `ProductRecyclerViewAdapter.OnProductClickListener` (línia 38). El tercer escoltador haurà sigut implementat anònimament a l'activitat i injectat a l'adaptador pel seu constructor.

Veiem la implementació anònima del tercer escoltador a l'activitat **`ShoppingActivity`** (línies 18-23):

```
1   public class ShoppingActivity extends AppCompatActivity {
2
3     /* Custom adapter for the recycler view of products */
4     private ProductRecyclerViewAdapter rvProductsAdapter;
5
6     ...
7
8     @Override
9     protected void onCreate(Bundle savedInstanceState) {
10      ...
11      initRecyclerView();
12    }
13
14    private void initRecyclerView() {
15      ...
16
17      rvProductsAdapter = new ProductRecyclerViewAdapter(
18          new ProductRecyclerViewAdapter.OnProductClickListener()
                {
19            @Override
20            public void OnProductClick(Product product) {
21              /* Llançar activitat amb detalls del producte */
22            }
23          }
24      );
25      binding.rvProducts.setAdapter(rvProductsAdapter);
26    }
27
28    ...
```

```
29 }
```

D'aquesta manera, finalment, haurem fet arribar el `Product` seleccionat de la vista reciclable, on podríem llançar una nova activitat per mostrar-ne els detalls (línia 21).

Podeu realitzar l'**Exercici 1**.

2.2 Observacions addicionals

En el codi base hi hem inclòs també un *widget* de tipus `SearchView`, el qual no hem cobert aquí. Si us interessa saber-ne més, podeu fer un cop d'ull a la Sec. A.

3 Exercicis

Exercici 1. Ara, es tractaria de d'afegir la funcionalitat d'esborrar elements de la vista reciclable mitjançant el clic d'un element d'UI que afegireu a `item_rv_product.xml`. No hauríeu de perdre la funcionalitat de mostrar els detalls. És a dir, si es fa clic a qualsevol lloc que no sigui el botó d'ocultació, es seguiran mostrant els detalls del producte. (Si no us queda clara la diferència, consulteu la Fig. 3.) Quan es detecti el clic al botó d'eliminar, l'adaptador haurà d'avisar a l'activitat de quin `Product` ha sigut clicat i, llavors, l'activitat demanar-li al model de vista d'esborrar el producte de la seva llista. Després, el model de vista haurà de confirmar l'eliminació a través d'un observable. L'activitat rebrà la confirmació i ho notificarà a l'adaptador per a que deixi de mostrar – ara sí – l'element. Fixeu-vos que és un doble camí dels esdeveniments en ambdós sentits: adaptador → activitat → model de vista i, després, model de vista → activitat → adaptador.

Passos a seguir:

1. Afegiu un `Button` (o `ImageView`) a la disposició dels ítems de la vista reciclable, `res/layout/item_rv_product.xml`, i assigneu-li l'identificador `btnRemoveItem` (o ivRemoveItem). Si voleu utilizar la mateixa icona de la Fig. 3, aneu a buscar el recurs `res/drawable/baseline_remove_circle_48.xml` i arrossegueu-lo a l'editor de la disposició com si es tractés d'un *widget*.
2. Definiu dues interfícies d'escoltador: (1) `OnPositionRemoveListener` a la classe `ProductViewHolder` amb un sol mètode de *callback* `void onPositionRemove(int position)` i (2) `OnProductRemoveListener` a l'adaptador amb un únic *callback* `void onProductRemove(Product product)`.
3. Al constructor de `ProductViewHolder`, afegiu-hi un paràmetre `OnPositionRemoveListener`.

4. En el mateix constructor, recupereu `btnRemoveItem` (o `ivRemoveItem`) de la disposició amb `findViewById` i setejeu-li un `View.OnClickListener` que en la seva implementació anònima cridi el *callback* de l'escoltador `OnPositionRemoveListener`, amb la posició del *view holder*.

5. Al constructor de l'adaptador, `ProductRecyclerViewAdapter`, passeu-li una instància del segon tipus d'escoltador que hem definit, `OnProductRemoveListener`, i guardeu-la en un atribut.

6. Aneu a l'`onCreateViewHolder(...)` i injecteu-li al `ProductViewHolder` una instància del `OnPositionRemoveListener` que en el seu *callback* que rebi la posició, la utilitzi per indexar `productList` i obtenir el `Product` i invocar el *callback* del `OnProductRemoveListener` passant-li el `Product`.

7. A `ShoppingActivity`, modifiqueu la inicialització de l'adaptador per passar-li una instància de l'`OnProductRemoveListener` que en el seu callback cridi `removeProduct` del `ShoppingViewModel` amb el `Product` rebut.

8. Examineu el mètode `void removeProduct(Product product)` de `ShoppingViewModel`, que ja us donem implementat, per veure a través de quin observable notifiquem l'eliminació i com.

9. A la `ShoppingActivity`, afegir la observació d'aquest observable i feu que amb l'emissió rebuda, cridi el mètode `void removeProductAt(int position)` de l'adaptador. Aquest mètode, ja implementat, farà `notifyItemRemoved(position)` [2] per treure – ara sí – el producte de la vista reciclable.

Trobareu marcades amb **Exercici 1** les parts on hi falti afegir codi.

Figure 3: Un clic a qualsevol lloc de l'element (excepte al `ImageView` amb el senyal de prohibit de color vermell i blanc) fa que es mostrin els detalls del producte. En canvi, un clic a l'`ImageView` farà que s'oculti l'ítem de la llista reciclable.

[2] És més eficient que `notifyDataSetChanged()` perquè, a diferència de l'anterior, no "repinta" tota la llista.

Appendix A SearchView

És un *widget* de tipus `SearchView` destinat a detectar entrada de l'usuari alhora que escriu. També proporciona diverses millores cosmètiques respecte un `EditText` i funcionals, amb la integració amb la barra d'accions (`ActionBar`) i proporcionant escoltadors diversos per a detectar que el *text ha canviat* o s'ha *entregat* (pulsació de `<ENTER>` o la lupa en el teclat virtual del dispositiu). Respectivament, `onQueryTextChange` o `onQueryTextSubmit`.

En la nostra aplicació, veureu que l'utilitzem per a "filtrar" el contingut de la vista reciclable de la `ShoppingActivity`. En el `onQueryTextSubmit` invoquem el mètode de delegació `fetchProductsByName(String queryText)` del model de vista `ShoppingViewModel` (enlloc del `fetchProductsCatalog()`, que carregaria tots els productes sense criteri). Aquest mètode demana fer la cerca a un servei que retorna al model de vista la llista de productes filtrats pel nom. Un cop disponible la llista en el model de vista, aquesta l'emet per l'observable per fer-la arribar a l'activitat.

Finalment, l'activitat utilitza la mateixa vista reciclable i adaptador per mostrar aquesta la llista de productes que per mostrar el catàleg sencer.

Us animem a traçar, vosaltres mateixos, aquest flux d'execució en el codi.

Trobareu més informació sobre `SearchView` aquí: `https://developer.android.com/reference/android/widget/SearchView`.