android

Laurent Provot <laurent.provot@uca.fr>

Octobre 2022

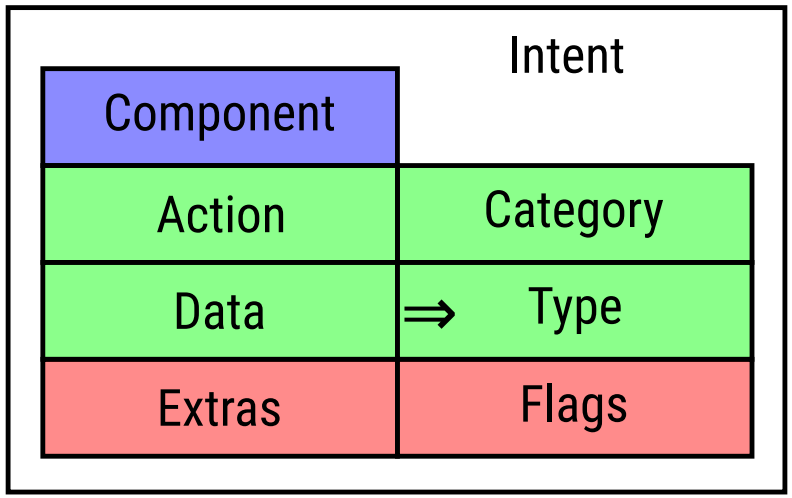
Dynamique au sein d'une application Android

Démarrage d'une activité

- On veut lancer une activité `B` depuis une activité `A`
  - Pas de `B()` directement dans `A`
  - On doit demander au système de le faire (permissions, cycle de vie, ...)
- `startActivity (intent: Intent)`
- `Intent` : décrit quoi lancer et comment

Les intentions

- Composant de communication au sein du système
- Agrégat d'informations diverses utiles au système pour faire des choix



- 2 types : explicite et implicite, suivant les infos renseignées
- En commun :
  - Extras : `Bundle` de données supplémentaires
  - Flags : comment l'activité est lancée

Les intentions

1. Intentions explicites (*Excplicit Intent*)
   - On décrit explicitement quel composant démarrer
     - `Intent(packageContext: Context, cls: Class<?>)`
       - `packageContext` : contexte du pkg applicatif de la classe
       - `cls` : méta-classe représentant l'activité à lancer
     - Propriété + setters (`component` , `setClass(…)` , `setClasseName(…)`)
   - L'activité à lancer doit être déclarée dans le Manifest sinon `ActivityNotFoundException`

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android" …>
    <application>
        …
        <activity android:name=".CheatActivity" … />
    </application>
</manifest>
```

Intention explicite : exemple

```
class A : Activity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        Button launchButton = findViewById<Button>(R.id.button_launch)
        launchButton.setOnClickListener {
                val launchIntent = Intent(this, B::class.java)
                startActivity(launchIntent)
        }
    }
}
```

Les intentions

2 Intentions implicites (*Implicit Intent*)

- On décrit l'action à faire et on laisse l'OS choisir les candidats
- Création d'un `Intent` « vide » puis customisation avec propriétés ou setters
- Infos portées par l'intention :
  - **Action** : le « truc » générique à faire `ACTION_VIEW`, `ACTION_SEND`, … (cf. classe `Intent` et propriété `action`)
  - **Data & Type** : URI vers la donnée et/ou type MIME (cf. `data`, `type`, `setDataAndType(…)`)
  - **Category** : info additionnelle sur l'action (`CATEGORY_BROWSABLE`, `CATEGORY_HOME`, …)
- `resolveActivity(…)` sur l'intent pour être sûr qu'il existe une activité qui l'accepte

Intention implicite : exemple

```
class A : Activity() {
    …
    fun composeEmail(addresses: String[], subject: String) {
        val mailIntent = Intent(Intent.ACTION_SENDTO)
        mailIntent.data = Uri.parse("mailto:")
        mailIntent.putExtra(Intent.EXTRA_EMAIL, addresses)
        mailIntent.putExtra(Intent.EXTRA_SUBJECT, subject)
        mailIntent.resolveActivity(packageManager)?.let {
            startActivity(mailIntent)
        }
    }
    …
}
```

Résolution des intentions impicites

- Grâce aux `<intent-filter>` déclarés dans le manifest
- Suivant 3 aspects : action, data, category

1. Test sur l'action

```
<intent-filter>
    <action android:name="android.intent.action.EDIT" />
    <action android:name="android.intent.action.VIEW" />
    …
</intent-filter>
```

- Pour passer le filtre, l'action déclarée dans l'intention doit correspondre à une action du filtre
- Un filtre sans action rejette toutes les intentions
- Une intention sans action déclarée passe le filtre (du moment qu'il contient au moins une action)

Résolution des intentions impicites

2. Test sur la catégorie

```
<intent-filter>
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    …
</intent-filter>
```

- Pour passer le filtre, chaque catégorie dans l'intention doit correspondre à une catégorie du filtre (réciproque pas vraie)
- Une intention sans catégorie déclarée passe le filtre, peu importe les catégories qu'il définit
- Le système ajoute automatiquement la catégorie `CATEGORY_DEFAULT` aux intentions implicites passées à `startActivity[ForResult](…)`
- Pour qu'une activité reçoive l'intention implicite il faut donc qu'elle déclare la catégorie `"android.intent.category.DEFAULT"`

Résolution des intentions impicites

2. Test sur la donnée (URI et type MIME)

```
<intent-filter>
    <data android:mimeType="video/mpeg" android:scheme="http" … />
    <data android:type="image/*" />
    …
</intent-filter>
```

- Les URI sont déclarées dans l'intention sous la forme `<scheme>://<host>:<port>/<path>`
- Les différents composants (et le(s) type(s) MIME) sont comparés à ceux déclarés dans le filtre pour savoir si l'`Intent` passe ou pas

Les intentions

3. Intentions en suspens (*Pending Intent*)

- `Intent` destinée à être lancée depuis un autre composant (`NotificationManager`, `AlarmManager` ou Home Screen)
- Opération spécifiée dans l'`Intent` effectuée par le composant comme s'il était l'application (même droits, même identité)
- Création d'un `Intent` (normalement explicite) puis encapsulation dans la `PendingIntent` grâce aux factory `PendingIntent.getActivity(…)`, `PendingIntent.getBroadcast(…)` ou `PendingIntent.getService()`

```
val intent = Intent(this, Target::class.java)
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
                Intent.FLAG_ACTIVITY_CLEAR_TASK)
pendingIntent = PendingIntent.getActivity(this, REQ_CD, intent, FLAGS)
```

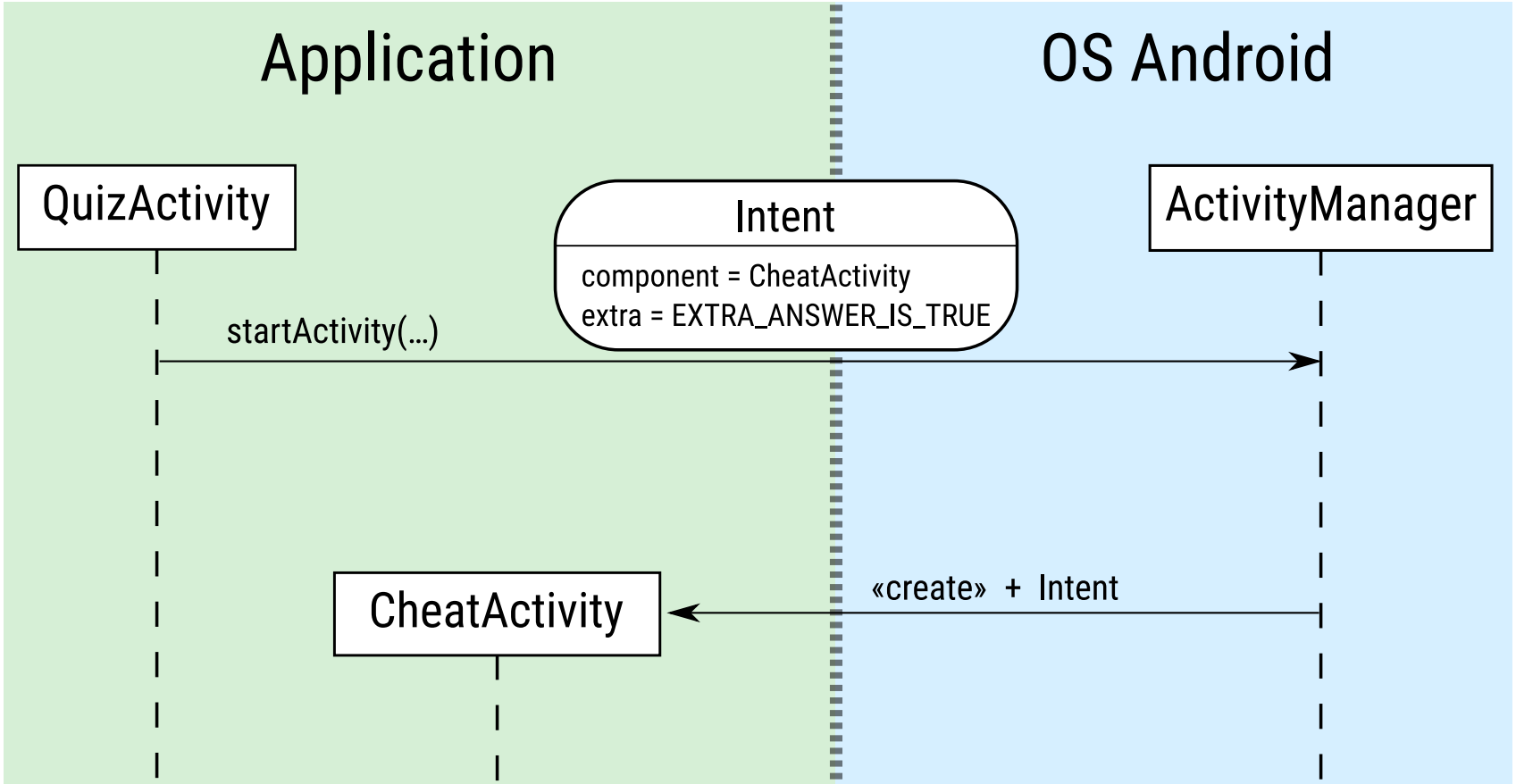
Passage d'arguments

- L'activité `B` à lancer a besoin de données en entrée
- Utilisation du bundle `extras` de l'intent
  `putExtra(key: String, value: XXX)`
  ou propriété `extras` + setters
- Bundle : table associative, clé → valeur
  - L'activité qui lance `B` doit connaître les clés ?
  - Découplage grâce à une *factory method*

```
class B : Activity() {
    companion object {
        private const val EXTRA_NAME_KEY = "org.pkg.name_key";

        fun newIntent(context: Context, x: XXX) =
            Intent(context, B::class.java).apply {
                putExtra(EXTRA_NAME_KEY, x)
            }
    }
    ...
```

Passage d'arguments : récapitulatif (Quiz)



Démarrage d'une activité

- Quid du cycle de vie des 2 activités `A` et `B` ?
- Il faut conserver une bonne expérience utilisateur
- Tant que `B` n'est pas prête à être interactive `A` reste visible

1. `A.onPause()`
2. `B.onCreate(…)` → `B.onStart()` → `B.onResume()`
3. `A.onStop()`

- Attention à ne pas mettre des opérations coûteuses dans ces méthodes

Données en retour d'une activité

- `B` ne connaît pas forcément l'activité `A` qui l'a lancée
- `B` ne doit pas créer une nouvelle instance de `A`
- Le même mécanisme de passage d'arguments ne peut donc pas être employé
- `A` devra préciser qu'elle lance `B` en espérant un résultat

DEPRECATED

- `startActivityForResult (`
  `intent: Intent, requestCode: Int): Unit`
  - `intent` : décrit quoi lancer
  - `requestCode` : associe un code à la demande
- `override fun onActivityResult (`
  `reqCode: Int, resultCode: Int, data: Intent)`

Données en retour d'une activité

- Utiliser la nouvelle Activity Result API
  - Meilleur découplage
  - Testabilité améliorée

1. Création d'un `ActivityResultLauncher` grâce à la méthode `registerForActivityResult(`
   `ct: ActivityResultContract,`
   `callback: ActivityResultCallback)`
   (définie dans l'interface `ActivityResultCaller`)
   - `ct` : le contract qui indique quoi faire
   - `callback` : la fonction qui sera utilisée pour traiter le résultat

- `StartActivityForResult()` comme valeur de `ct` aura le même effet que l'ancien `startActivityForResult()`

Données en retour d'une activité

2. Lancement de `B` grâce à la méthode `launch()` de `ActivityResultLauncher`
   - `A` doit être au moins dans l'état *« Created »*
3. `B` spécifiera les informations à retourner à travers une intent

- `setResult (resultCode: Int, data: Intent): Unit`
  - `resultCode` : entier décrivant l'issue de l'action `RESULT_OK`, `RESULT_CANCELED`, `RESULT_FIRST_USER`
  - `data` : intent pour passer des données en retour

4. Au retour dans `A` la `callback` définie lors du `registerForActivityResult()` est appelée

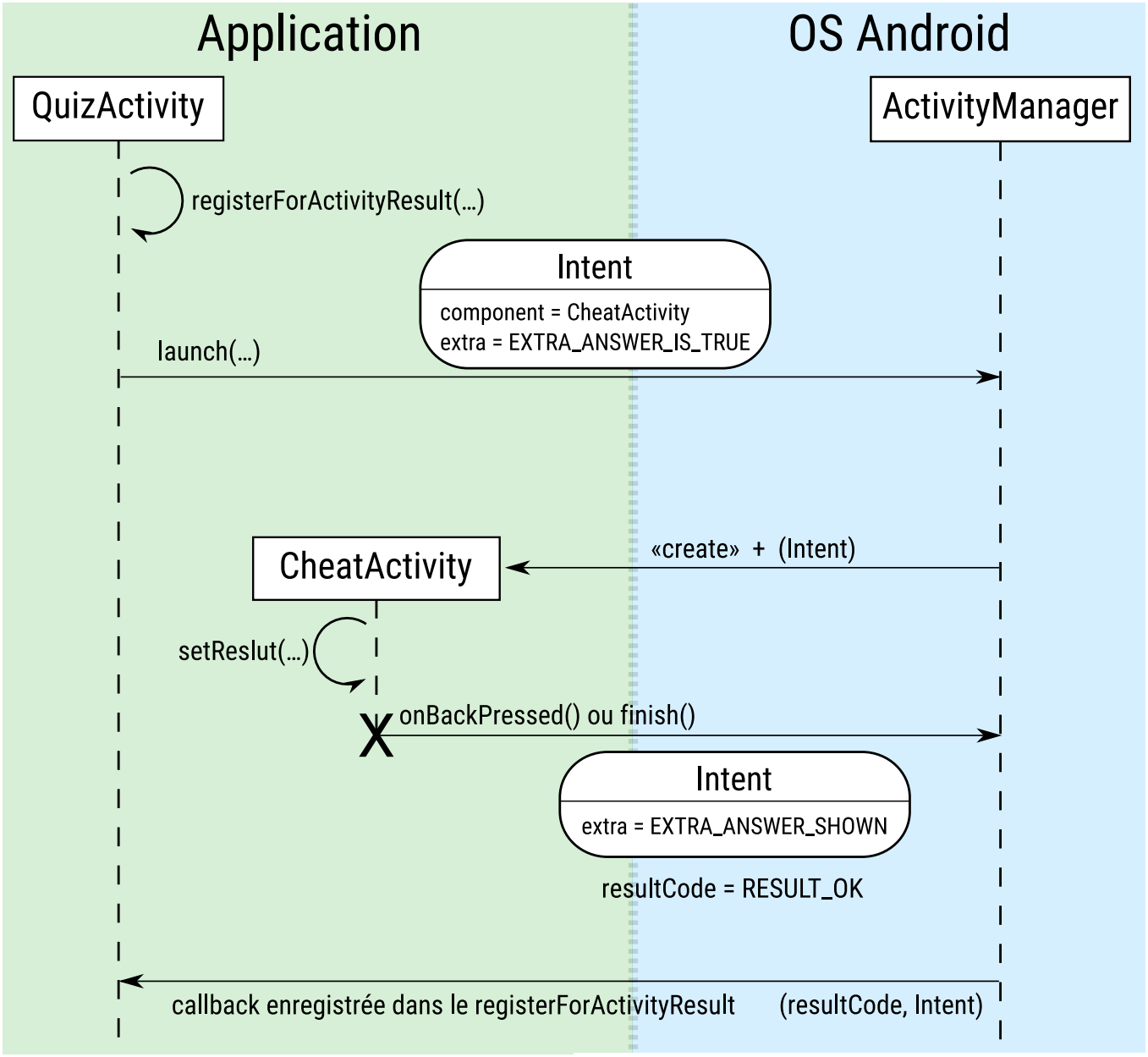
Données en retour : exemple (Quiz)

```
// Création du launcher (dans le Ctor par exemple)
private val cheatLauncher = registerForActivityResult(
                                StartActivityForResult()) { result ->
  if (result.resultCode == Activity.RESULT_OK)
    result.data?.let { isCheater = CheatActivity.wasAnswerShown(it) }
}

// Lancement de l'activité de triche
findViewById<Button>(R.id.button_cheat).setOnClickListener {
  cheatLauncher.launch(CheatActivity.newIntent(this,
                       questions[currentQuestionIndex].isAnswerTrue))
}

// Dans CheatActivity
private fun setHasCheated() {
  val data = Intent().apply { putExtra(EXTRA_ANSWER_SHOWN, true) }
  setResult(Activity.RESULT_OK, data)
}
```

Données en retour : récapitulatif (Quiz)



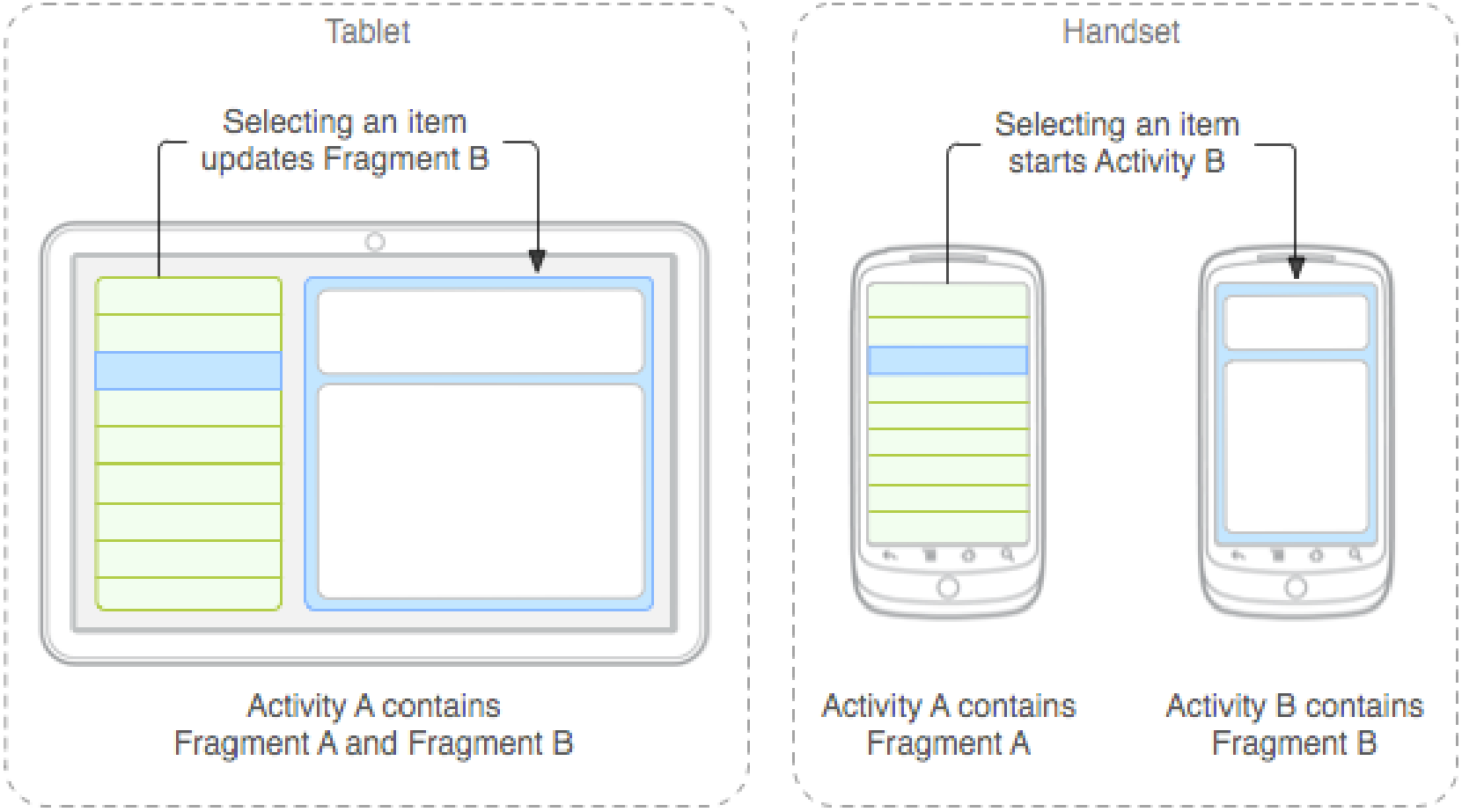
Données en retour d'une activité

- la `callback` est appelée juste avant `onResume()`
- `setResult(…)` doit être appelé avant un appel à `finish()` sinon `resultCode = RESULT_CANCELED` et `data = null`
- Attention `onBackPressed()` appelle par défaut `finish()`
- Du coup `setResult(…)` dans `onPause()` : c'est trop tard !

Master-Detail et Fragments

Master-Detail et Fragments

- Préserver l'ergonomie pour différentes tailles d'écran



Fragment

- But : séparer les composants d'une activité (complexe) en sous-composants autonomes
- Apparu avec HoneyComb (API v11)
- Un fragment = (en quelque sorte) une sous-activité
- Un fragment doit être lié avec une activité pour être utile
- Un fragment ne doit pas être dépendant de l'activité qui le contient (découplage)
- Un fragment possède ses propres méthodes du cycle de vie

Fragments

- 2 possibilités pour utiliser les fragments

1. Utiliser les fragments natifs :
   - Dispo de base dans la classe `Activity` si $>$ API v11
   - Profite des dernières avancées du framework
   - Pas de support pour les appareils pré Andoid HoneyComb
2. Utiliser *AndroidX* (anciennement la *Support Library*) :
   - Dériver de `FragmentActivity` (ou `AppCompatActivity`)
   - Utiliser certaines propriétés ou méthodes « alternatives » `supportFragmentManager`, `supportActionBar` et `xmlns:app` lors du design
   - Backport pour les anciennes versions d'Android (jusqu'à v4)
   - Mises à jour régulières
   - En prime : `Toolbar`, Material Design, ...

- Conseil : utiliser *AndroidX* !!!

Cycle de vie Fragment vs Activity

Cf. Cycle de vie complet

Ajout d'un fragment statique

- Implémentation rapide
- Design figé, on ne peut pas changer de fragment

1. Au niveau du fragment : création de la vue dans le `onCreateView(…)`

```
package fr.uca.iut
class MyFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater,
                              containter: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        val view = inflater.inflate(R.layout.fragment_layout,
                                    container, false)
        // configuration de la vue (findViewById, …)
        return view
    }
}
```

Ajout d'un fragment statique

2. Au niveau de l'activité : ajout d'un élément `fragment` dans le layout (ou bien `FragmentContainerView`)

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android" … >
    <fragment
        android:id="@+id/id_fragment"
        android:name="fr.iut.MyFragment" />
</LinearLayout>
```

3. Il y a juste à lier l'activité à son layout

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    // on pourra ensuite récupérer si nécessaire le fragment
    fragment = fragmentManager.findFragmentById(R.id.id_fragment)
}
```

Ajout d'un fragment dynamiquement

- On utilise le `FragmentManager`
- Il opère des transactions pour ajouter/remplacer/supprimer les fragments
- Il permet de gérer la backstack des fragments (gestion non automatique)

1. Au niveau de `MyFragment` rien ne change, la vue est toujours créée dans le `onCreateView(…)`
2. Au niveau de l'activité : le layout contient un « placeholder » pour le fragment (généralement un `FrameLayout`)

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android" … >
    <FrameLayout
        android:id="@+id/id_fragment" />
</LinearLayout>
```

Ajout d'un fragment dynamiquement

3. Il faut lier l'activité à son layout et charger le fragment manuellement

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    if (fragmentManager.findFragmentById(R.id.id_fragment) == null) {
        fragmentManager.beginTransaction()
                       .add(R.id.id_fragment, MyFragment())
                       .commit()
    }
}
```

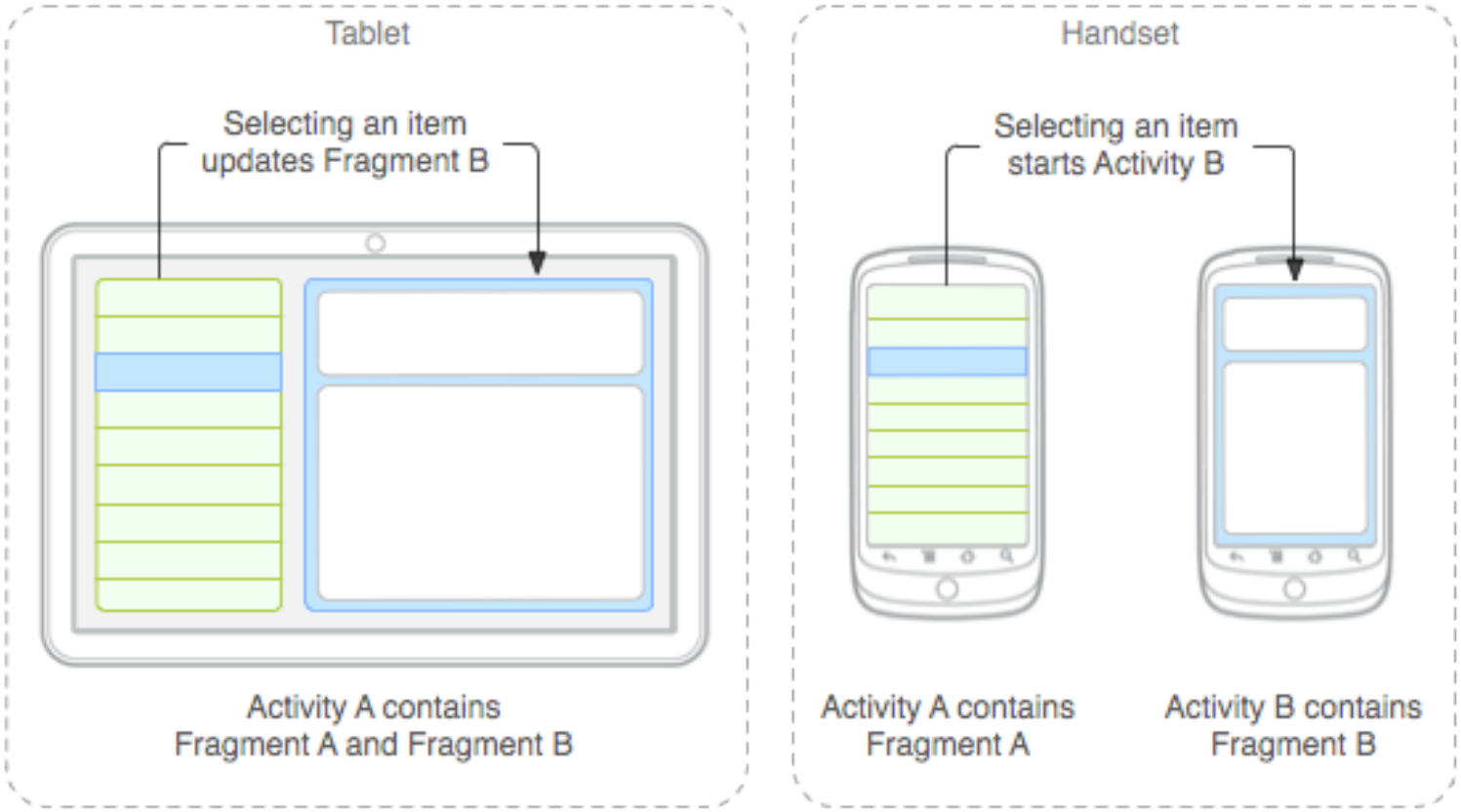
Backstack des fragments

- L'`ActivityManager` gère la backstack des activités automatiquement
- Pour les fragments il faut dire explicitement au `FragmentManager` de le faire

```
val ft = getFragmentManager().beginTransaction()
// gestion de la transaction
ft.addToBackStack("Tag transaction")
ft.commit()
```

```
override fun onBackPressed() {
    if (fragmentManager.backStackEntryCount > 0) {
        fragmentManager.popBackStack()
    } else {
        super.onBackPressed()
    }
}
```

Retour au Master Detail



- Gérer la vue de type liste
- Gérer les activités / fragments suivant le type d'affichage

RecyclerView

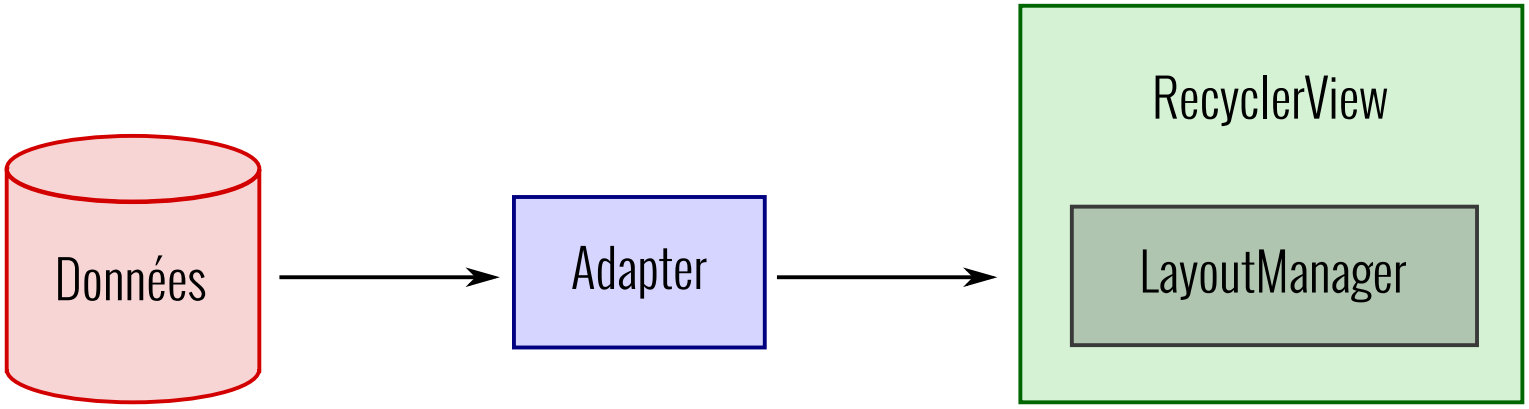
Une liste écolo

- On veut afficher des données (un grand nombre)
- Il faut définir « comment » les afficher
- Il faut en afficher seulement un sous-ensemble restreint à la fois
- Il faut conserver une certaine efficacité
- Il faut rester évolutif

- C'est le rôle de la `RecyclerView` (*AndroidX*)

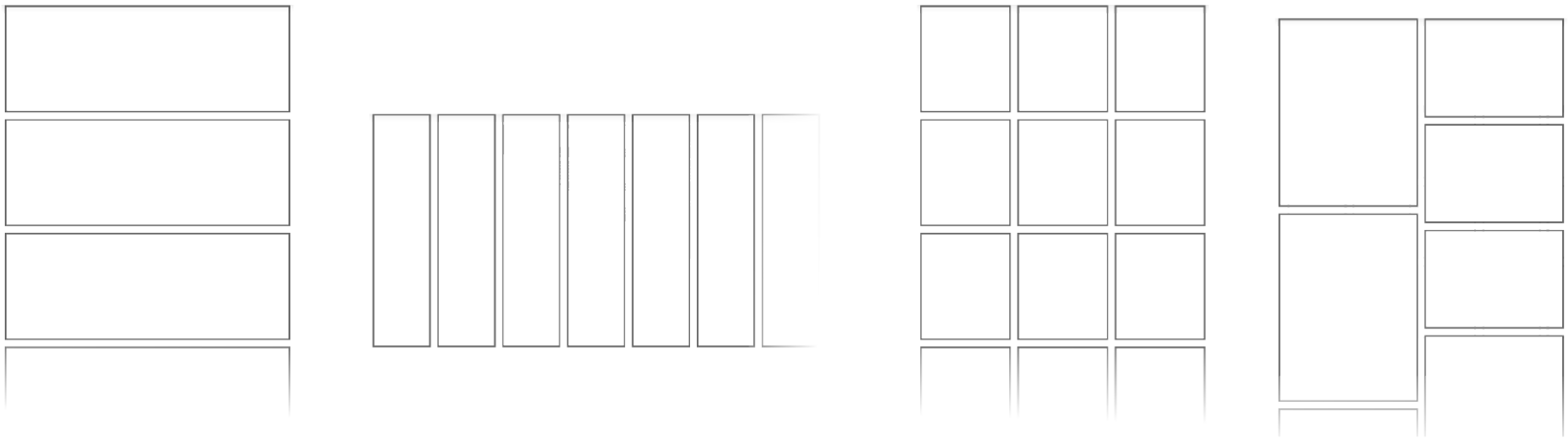
Lien avec le modèle

- Articulation de la `RecyclerView`



- La `RecyclerView` décide combien d'éléments afficher
- Le `LayoutManager` gère l'agencement des vues des éléments
- L'`Adapter` permet d'obtenir des vues en fonction des données

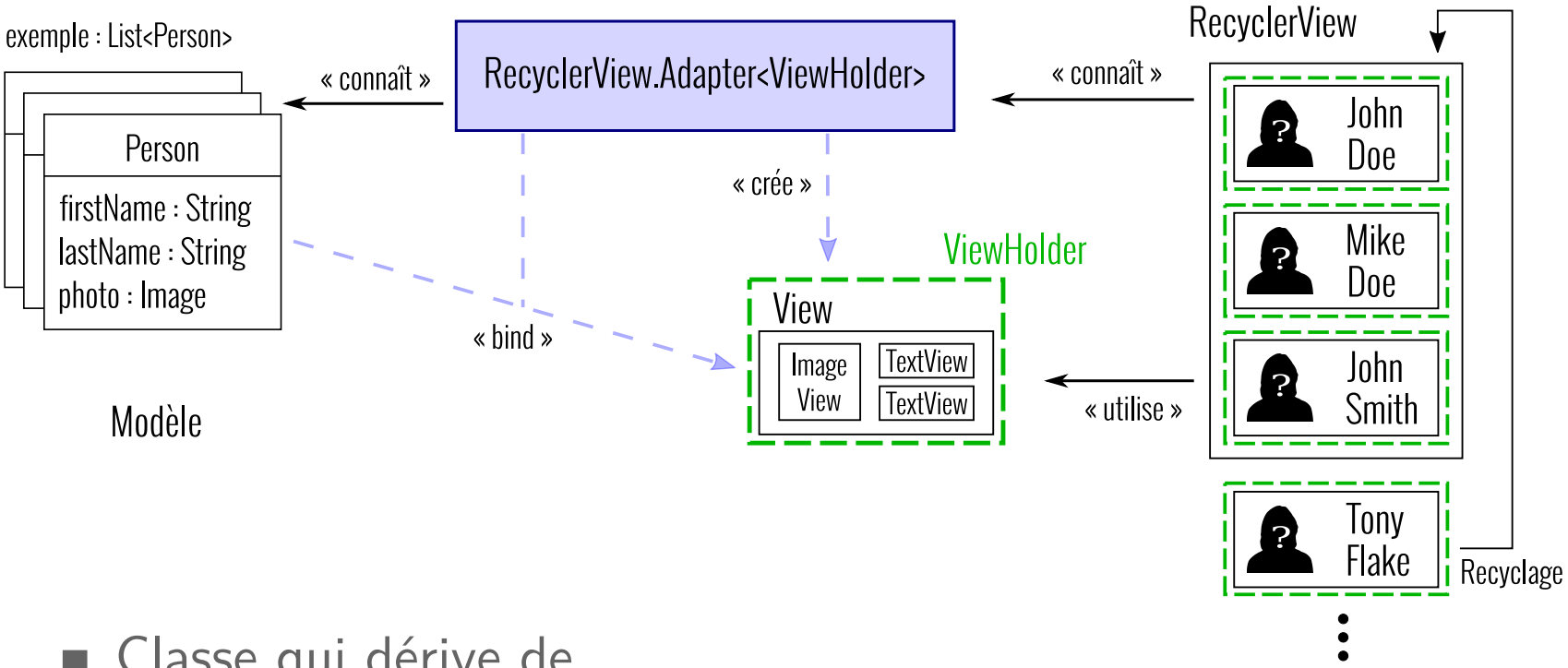
L'agencement des vues



- Classes dérivées de `RecyclerView.LayoutManager`
- De base : `LinearLayoutManager`, `GridLayoutManager`, `StaggeredGridLayoutManager`

```
mRecyclerView = findViewById<RecyclerView>(R.id.my_recycler_view)
mRecyclerView.layoutManager = LinearLayoutManager(/* context */)
// optimisation si la taille de la RecyclerView ne change pas
mRecyclerView.setHasFixedSize(true);
```

Adaptation des données



- Classe qui dérive de `RecyclerVew.Adapter<RecyclerView.ViewHolder>`
  - Connaître la donnée à afficher
  - Pouvoir créer les vues
  - Être efficace avec beaucoup d'éléments

Méthodes à redéfinir

- `fun getItemCount(): Int`
  permet à la `RecyclerView` de savoir combien d'éléments elle va devoir gérer
- `fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder`
  permet de créer un `ViewHolder` qui gère une vue d'un élément
- `fun onBindViewHolder(holder: ViewHolder, position: Int)`
  permet de lier la donnée métier à sa vue; leur relation est leur position (index)
- À quoi sert le `ViewHolder` ? Efficacité (car ré-instanciation de vue et `findViewById(…)` coûteux)

Lien entre le Master et le Detail

1. Les activités agencent les fragments
2. Les fragments sont indépendants de l'activité

- Comment l'information est-elle passée entre un fragment et une activité ?
- Clic sur un élément de liste : lancement d'activité ou remplacement de fragment ?

Passage d'arguments

- Le fragment est lié à une activité : propriété `activity`
- On pourrait utiliser `activity.intent` ?
- Pour un vrai découplage les fragments possèdent leur propres arguments
- Bundle géré par la propriété `arguments`
- Même principe que pour le lancement d'activité → *factory method* pour l'instanciation du fragment :

```
companion object {
    private const val PARAM_KEY = "org.pkg.param_key"

    fun newInstance(param: XXX) = MyFragment().apply {
        arguments = Bundle().apply {
            putXXX(PARAM_KEY, param)
        }
}
```

Communication entre le fragment et son activité

- On veut conserver une indépendance du fragment
- Comme le fragment ne connait pas son contexte, il notifie juste des interactions en son sein
- Principe des *listeners* en Java
- On définit une interface pour les échanges
- L'activité implémente cette interface

```
class MyFragment : Fragment() {
    interface OnSomethingListener {
        fun onSomething(param: XXX)
    }
    private val listener: OnSomethingListener = …
}

class MyActivity : Activity(), MyFragment.OnSomethingListener {
    override fun onSomething(param: XXX) { … }
}
```