Troisième partie

Vers des applications plus complexes

13. Préambule : quelques concepts avancés

Le Manifest est un fichier que vous trouverez à la racine de votre projet sous le nom d'`Android Manifest.xml` et qui vous permettra de spécifier différentes options pour vos projets, comme le matériel nécessaire pour les faire fonctionner, certains paramètres de sécurité ou encore des informations plus ou moins triviales telles que le nom de l'application ainsi que son icône.

Mais ce n'est pas tout, c'est aussi la première étape à maîtriser afin de pouvoir insérer plusieurs activités au sein d'une même application, ce qui sera la finalité des deux prochains chapitres.

Ce chapitre se chargera aussi de vous expliquer plus en détail comment manipuler le cycle d'une activité. En effet, pour l'instant nous avons toujours tout inséré dans `onCreate`, mais il existe des situations pour lesquelles cette attitude n'est pas du tout la meilleure à adopter.

13.1. Généralités sur le nœud <manifest>

Ce fichier est indispensable pour tous les projets Android, c'est pourquoi il est créé par défaut. Si je crée un nouveau projet, voici le Manifest qui est généré :

```
```

Voyons un petit peu de quoi il s'agit ici.

13.1.1. <manifest>

La racine du Manifest est un nœud de type `<manifest>`. Comme pour les vues et les autres ressources, on commence par montrer qu'on utilise l'espace de noms `android` :

```
```

Puis, on déclare dans quel package se trouve notre application :

```
```

… afin de pouvoir utiliser directement les classes qui se situent dans ce package sans avoir à préciser à chaque fois qu'elles s'y situent. Par exemple, dans notre Manifest actuel, vous pouvez voir la ligne suivante : `android:name=".ManifestActivity"`. Elle fait référence à l'activité principale de mon projet : `ManifestActivity`. Cependant, si nous n'avions

pas précisé `package="sdz.chapitreTrois"`, alors le nœud `android:name` aurait dû valoir `android:name="sdz.chapitreTrois.ManifestActivity"`. Imaginez seulement que nous ayons à le faire pour chaque activité de notre application… Cela risquerait d'être vite usant.

N'oubliez pas que le nom de package doit être unique si vous voulez être publié sur le Play Store. De plus, une fois votre application diffusée, ne changez pas ce nom puisqu'il agira comme un identifiant unique pour votre application.

Toujours dans le nœud `<manifest>`, il est ensuite possible d'indiquer quelle est la version actuelle du logiciel :

```

```

L'attribut `android:versionCode` doit être un nombre entier (positif et sans virgule) qui indique quelle est la version actuelle de l'application. Mais attention, il ne s'agit pas du nombre qui sera montré à l'utilisateur, juste celui considéré par le Play Store. Si vous soumettez votre application avec un code de version supérieur à celui de votre ancienne soumission, alors le Play Store saura que l'application a été mise à jour. En revanche, le `android:versionName` peut être n'importe quelle chaîne de caractères et sera montré à l'utilisateur. Rien ne vous empêche donc de mettre `android:versionName="Première version alpha - 0.01a"` par exemple.

13.1.2. <uses-sdk>

On utilise ce nœud de manière à pouvoir filtrer les périphériques sur lesquels l'application est censée fonctionner en fonction de leur version d'Android. Ainsi, il vous est possible d'indiquer la version minimale de l'API que doit utiliser le périphérique :

```

```

Ici, il faudra la version 2.1 d'Android (API 7) ou supérieure pour pouvoir utiliser cette application.

Votre application ne sera proposée sur le Google Play qu'à partir du moment où l'utilisateur utilise cette version d'Android ou une version supérieure.

Il existe aussi un attribut `android:targetSdkVersion` qui désigne non pas la version minimale d'Android demandée, mais plutôt la version à partir de laquelle on pourra exploiter à fond l'application. Ainsi, vous avez peut-être implémenté des fonctionnalités qui ne sont disponibles qu'à partir de versions d'Android plus récentes que la version minimale renseignée avec `android:minSdkVersion`, tout en faisant en sorte que l'application soit fonctionnelle en utilisant une version d'Android égale ou supérieure à celle précisée dans `android:minSdkVersion`.

Vous pouvez aussi préciser une limite maximale à respecter avec `android:maxSdkVersion` si vous savez que votre application ne fonctionne pas sur les plateformes les plus récentes, mais je ne vous le conseille pas, essayez plutôt de rendre votre application compatible avec le plus grand nombre de terminaux !

13.1.3. <uses-feature>

Étant donné qu'Android est destiné à une très grande variété de terminaux différents, il fallait un moyen pour faire en sorte que les applications qui utilisent certains aspects hardware ne puissent être proposées que pour les téléphones qui possèdent ces capacités techniques. Par exemple, vous n'allez pas proposer un logiciel pour faire des photographies à un téléphone qui ne possède pas d'appareil photo (même si c'est rare) ou de capture sonore pour une tablette qui n'a pas de microphone (ce qui est déjà moins rare).

Ce nœud peut prendre trois attributs, mais je n'en présenterai que deux :

13.1.3.1. android :name

Vous pouvez préciser le nom du matériel avec l'attribut `android:name`. Par exemple, pour l'appareil photo (et donc la caméra), on mettra :

Cependant, il arrive qu'on ne cherche pas uniquement un composant particulier mais une fonctionnalité de ce composant ; par exemple pour permettre à l'application de n'être utilisée que sur les périphériques qui ont un appareil photo avec autofocus, on utilisera :

Vous trouverez sur cette page la liste exhaustive des valeurs que peut prendre `android:name`.

13.1.3.2. android :required

Comme cet attribut n'accepte qu'un booléen, il ne peut prendre que deux valeurs :

1. `true` : ce composant est indispensable pour l'utilisation de l'application.
2. `false` : ce composant n'est pas indispensable, mais sa présence est recommandée.

13.1.4. <supports-screens>

Celui-ci est très important, il vous permet de définir quels types d'écran supportent votre application. Cet attribut se présente ainsi :

```
```

Si un écran est considéré comme petit, alors il entrera dans la catégorie `smallScreen`, s'il est moyen, c'est un `normalScreen`, et les grands écrans sont des `largeScreen`.

L'API 9 a vu l'apparition de l'attribut `<supports-screens android:xlargeScreens="boolean" />`. Si cet attribut n'est pas défini, alors votre application sera lancée avec un mode de compatibilité qui agrandira tous les éléments graphiques de votre application, un peu comme si on faisait un zoom sur une image. Le résultat sera plus laid que si vous développiez une interface graphique dédiée, mais au moins votre application fonctionnera.

</supports-screens></uses-feature></uses-sdk></manifest>

13.2. Le nœud <application>

Le nœud le plus important peut-être. Il décrit les attributs qui caractérisent votre application et en énumère les composants de votre application. Par défaut, votre application n'a qu'un composant, l'activité principale. Mais voyons d'abord les attributs de ce nœud.

Vous pouvez définir l'icône de votre application avec `android:icon`, pour cela vous devez faire référence à une ressource drawable : `android:icon="@drawable/ic_launcher"`.

Ne confondez pas avec `android:logo` qui, lui, permet de changer l'icône *uniquement* dans l'ActionBar.

Il existe aussi un attribut `android:label` qui permet de définir le nom de notre application.

13.2.1. Les thèmes

Vous savez déjà comment appliquer un style à plusieurs vues pour qu'elles respectent les mêmes attributs. Et si nous voulions que toutes nos vues respectent un même style au sein d'une application ? Que les textes de toutes ces vues restent noirs par exemple ! Ce serait contraignant d'appliquer le style à chaque vue. C'est pourquoi il est possible d'appliquer un style à une application, auquel cas on l'appelle un **thème**. Cette opération se déroule dans le Manifest, il vous suffit d'insérer l'attribut :

Vous pouvez aussi exploiter les thèmes par défaut fournis par Android, par exemple pour que votre application ressemble à une boîte de dialogue :

Vous en retrouverez d'autres sur la documentation .

Laissez-moi maintenant vous parler de la notion de composants. Ce sont les éléments qui composeront vos projets. Il en existe cinq types, mais vous ne connaissez pour l'instant que les activités, alors je ne vais pas vous embrouiller plus avec ça. Sachez juste que votre application sera au final un ensemble de composants qui interagissent, entre eux et avec le reste du système.

13.2.2. <activity>

Ce nœud permet de décrire toutes les activités contenues dans notre application. Comme je vous l'ai déjà dit, une activité correspond à un écran de votre application, donc, si vous voulez avoir plusieurs écrans, il vous faudra plusieurs activités.

Le seul attribut vraiment indispensable ici est `android:name`, qui indique quelle est la classe qui implémente l'activité.

`android:name` définit aussi un identifiant pour Android qui permet de repérer ce composant parmi tous. Ainsi, ne changez pas l'attribut `android:name` d'un composant au cours d'une mise à jour, sinon vous risquez de rencontrer des effets de bord assez désastreux.

Vous pouvez aussi préciser un nom pour chaque activité avec `android:label`, c'est le mot qui s'affichera en haut de l'écran sur notre activité. Si vous ne le faites pas, c'est la `String` renseignée dans le `android:label` du nœud `<application>` qui sera utilisée.

Vous pouvez voir un autre nœud de type `<intent-filter>` qui indique comment se lancera cette activité. Pour l'instant, sachez juste que l'activité qui sera lancée depuis le menu principal d'Android contiendra toujours dans son Manifest ces lignes-ci :

Je vous donnerai beaucoup plus de détails dans le prochain chapitre.

Vous pouvez aussi définir un thème pour une activité, comme nous l'avons fait pour une application.</activity>

13.3. Les permissions

Vous le savez sûrement, quand vous téléchargez une application sur le Play Store, on vous propose de regarder les autorisations que demande cette application avant de commencer le téléchargement (voir figure suivante). Par exemple, pour une application qui vous permet de retenir votre numéro de carte bancaire, on peut légitimement se poser la question de savoir si dans ses autorisations se trouve « Accès à internet ».

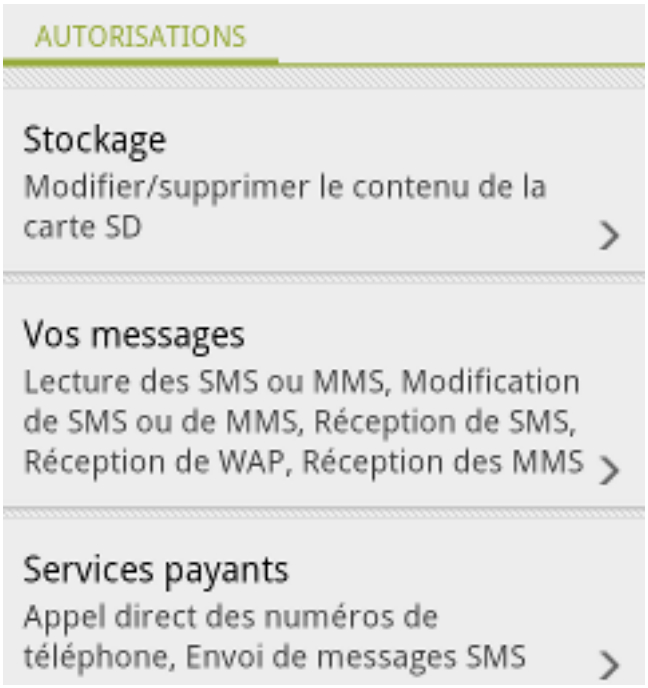

FIGURE 13.1. – On vous montre les autorisations que demande l'application avant de la télécharger

Par défaut, aucune application ne peut exécuter d'opération qui puisse nuire aux autres applications, au système d'exploitation ou à l'utilisateur. Cependant, Android est constitué de manière à ce que les applications puissent partager. C'est le rôle des permissions, elles permettent de limiter l'accès aux composants de vos applications.

13.3.1. Utiliser les permissions

Afin de pouvoir utiliser certaines API d'Android, comme l'accès à internet dans le cas précédent, vous devez préciser dans le Manifest que vous utilisez les permissions. Ainsi, l'utilisateur final est averti de ce que vous souhaitez faire, c'est une mesure de protection importante à laquelle vous devez vous soumettre.

Vous trouverez sur cette page une liste des permissions qui existent déjà.

Ainsi, pour demander un accès à internet, on indiquera la ligne :

13.3.2. Créer ses permissions

Il est important que votre application puisse elle aussi partager ses composants puisque c'est comme ça qu'elle sait se rendre indispensable.

Une permission doit être de cette forme-ci :

```
```

Où :

- `android:name` est le nom qui sera utilisé dans un `uses-permission` pour faire référence à cette permission. Ce nom doit être unique.
- `android:label` est le nom qui sera indiqué à l'utilisateur, faites-en donc un assez explicite.
- `android:description` est une description plus complète de cette permission que le `label`.
- `android:icon` est assez explicite, il s'agit d'une icône qui est censée représenter la permission. Cet attribut est heureusement facultatif.
- `android:protectionLevel` indique le degré de risque du composant lié à cette permission. Il existe principalement trois valeurs :
  - `normal` si le composant est sûr et ne risque pas de dégrader le comportement du téléphone ;
  - `dangerous` si le composant a accès à des données sensibles ou peut dégrader le fonctionnement du périphérique ;
  - `signature` pour n'autoriser l'utilisation du composant que par les produits du même auteur (concrètement, pour qu'une application se lance sur votre terminal ou sur l'émulateur, il faut que votre application soit « approuvée » par Android. Pour ce faire, un certificat est généré — même si vous ne le demandez pas, l'ADT s'en occupe automatiquement — et il faut que les certificats des deux applications soient identiques pour que la permission soit acceptée).

13.4. Gérer correctement le cycle des activités

Comme nous l'avons vu, quand un utilisateur manipule votre application, la quitte pour une autre ou y revient, elle traverse plusieurs états symbolisés par le cycle de vie des activités, schématisé à la figure suivante.

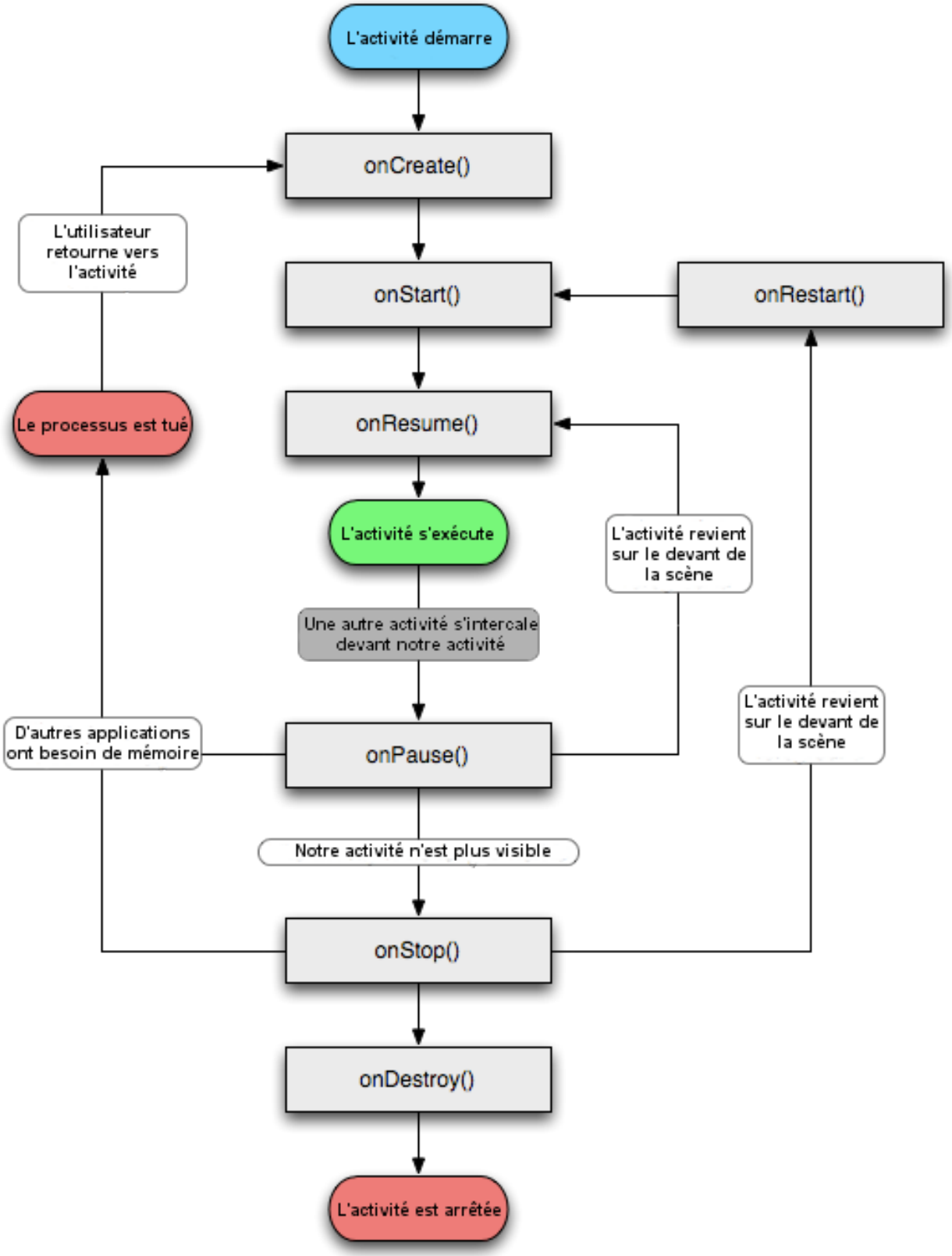

FIGURE 13.2. – Le cycle de vie d'une activité

La transition entre chaque étape implique que votre application appelle une méthode. Cette

méthode partage le même nom que l'état traversé, par exemple à la création est appelée la méthode `onCreate`.

Ces méthodes ne sont que des états de transition très éphémères entre les trois grands états dont nous avons déjà discuté : la période **active** peut être interrompue par la période **suspendue**, qui elle aussi peut être interrompue par la période **arrêtée**.

Vous le comprendrez très vite, l'entrée dans chaque état est symbolisée par une méthode et la sortie de chaque état est symbolisée par une autre méthode. Ces deux méthodes sont complémentaires : ce qui est initialisé dans la première méthode doit être stoppé dans la deuxième, et ce presque toujours.

13.4.1. Sous les feux de la rampe : période suspendue

Cette période débute avec `onResume()` et se termine avec `onPause()`. On entre dans la période suspendue dès que votre activité n'est plus que partiellement visible, comme quand elle est voilée par une boîte de dialogue. Comme cette période arrive fréquemment, il faut que le contenu de ces deux méthodes s'exécute rapidement et nécessite peu de processeur.

13.4.1.1. onPause()

On utilise la méthode `onPause()` pour arrêter des animations, libérer des ressources telles que le GPS ou la caméra, arrêter des tâches en arrière-plan et de manière générale stopper toute activité qui pourrait solliciter le processeur. Attention, on évite de sauvegarder dans la base de données au sein de `onPause()`, car c'est une action qui prend beaucoup de temps à se faire, et il vaut mieux que `onPause()` s'exécute rapidement pour fluidifier la manipulation par l'utilisateur. De manière générale, il n'est pas nécessaire de sauvegarder des données dans cette méthode, puisque Android conserve une copie fonctionnelle de l'activité, et qu'au retour elle sera restaurée telle quelle.

13.4.1.2. onResume()

Cette méthode est exécutée à chaque fois que notre activité retourne au premier plan, mais aussi à chaque lancement, c'est pourquoi on l'utilise pour initialiser les ressources qui seront coupées dans le `onPause()`. Par exemple, dans votre application de localisation GPS, vous allez initialiser le GPS à la création de l'activité, mais pas dans le `onCreate(Bundle)`, plutôt dans le `onResume()` puisque vous allez le couper à chaque fois que vous passez dans le `onPause()`.

13.4.2. Convoquer le plan et l'arrière-plan : période arrêtée

Cette fois-ci, votre activité n'est plus visible du tout, mais elle n'est pas arrêtée non plus. C'est le cas si l'utilisateur passe de votre application à une autre (par exemple s'il retourne sur l'écran d'accueil), alors l'activité en cours se trouvera stoppée et on reprendra avec cette activité dès que

l'utilisateur retournera dans l'application. Il est aussi probable que dans votre application vous ayez plus d'une activité, et passer d'une activité à l'autre implique que l'ancienne s'arrête.

Cet état est délimité par `onStop()` (toujours précédé de `onPause()`) et `onRestart()` (toujours suivi de `onResume()`, puis `onStart()`)). Cependant, il se peut que l'application soit tuée par Android s'il a besoin de mémoire, auquel cas, après `onStop()`, l'application est arrêtée et, quand elle sera redémarrée, on reprendra à `onCreate(Bundle)`.

Là, en revanche, vous devriez sauvegarder les éléments dans votre base de données dans `onStop()` et les restaurer lorsque c'est nécessaire (dans `onStart()` si la restauration doit se faire au démarrage et après un `onStop()` ou dans `onResume()` si la restauration ne doit se faire qu'après un `onStop()`).

13.4.3. De la naissance à la mort

13.4.3.1. onCreate(Bundle)

Première méthode qui est lancée au démarrage de l'activité, c'est l'endroit privilégié pour initialiser l'interface graphique, pour démarrer les tâches d'arrière-plan qui s'exécuteront pendant toute la durée de vie de l'activité, pour récupérer des éléments de la base de données, etc.

Il se peut très bien que vous utilisiez une activité uniquement pour faire des calculs et prendre des décisions entre l'exécution de deux activités, auquel cas vous pouvez faire appel à la méthode `public void finish ()` pour passer directement à la méthode `onDestroy()`, qui symbolise la mort de l'activité. Notez bien qu'il s'agit du seul cas où il est recommandé d'utiliser la méthode `finish()` (c'est-à-dire qu'on évite d'ajouter un bouton pour arrêter son application par exemple).

13.4.3.2. onDestroy()

Il existe trois raisons pour lesquelles votre application peut atteindre la méthode `onDestroy`, c'est-à-dire pour lesquelles on va terminer notre application :

— Comme je l'ai déjà expliqué, il peut arriver que le téléphone manque de mémoire et qu'il ait besoin d'arrêter votre application pour en récupérer.
— Si l'utilisateur presse le bouton `Arrière` et que l'activité actuelle ne permet pas de retourner à l'activité précédente (ou s'il n'y en a pas !), alors on va quitter l'application.
— Enfin, si vous faites appel à la méthode `public void finish()` — mais on évite de l'utiliser en général —, il vaut mieux laisser l'utilisateur appuyer sur le bouton `Arrière`, parce qu'Android est conçu pour gérer le cycle des activités tout seul (c'est d'ailleurs la raison pour laquelle il faut éviter d'utiliser des *task killers*).

Comme vous le savez, c'est dans `onCreate(Bundle)` que doivent être effectuées les différentes initialisations ainsi que les tâches d'arrière-plan qu'on souhaite voir s'exécuter pendant toute l'application. Normalement, quand l'application arrive au `onDestroy()`, elle est déjà passée par `onPause()` et `onStop()`, donc la majorité des tâches de fond auront été arrêtées ; cependant, s'il s'agit d'une tâche qui devrait s'exécuter pendant toute la vie de l'activité — qui aura été démarrée dans `onCreate(Bundle)` —, alors c'est dans `onDestroy()` qu'il faudra l'arrêter.

Android passera d'abord par `onPause()` et `onStop()` dans tous les cas, à l'exception de l'éventualité où vous appeleriez la méthode `finish()` ! Si c'est le cas, on passe directement au `onDestroy()`. Heureusement, il vous est possible de savoir si le `onDestroy()` a été appelé suite à un `finish()` avec la méthode `public boolean isFinishing()`.

Si à un moment quelconque votre application lance une exception que vous ne catchez pas, alors l'application sera détruite sans passer par le `onDestroy()`, c'est pourquoi cette méthode n'est pas un endroit privilégié pour sauvegarder des données.

13.4.4. L'échange équivalent

Quand votre application est quittée de manière normale, par exemple si l'utilisateur presse le bouton `Arrière` ou qu'elle est encore ouverte et que l'utilisateur ne l'a plus consultée depuis longtemps, alors Android ne garde pas en mémoire de traces de vos activités, puisque l'application s'est arrêtée correctement. En revanche, si Android a dû tuer le processus, alors il va garder en mémoire une trace de vos activités afin de pouvoir les restaurer telles quelles. Ainsi, au prochain lancement de l'application, le paramètre de type `Bundle` de la méthode `onCreate(Bundle)` sera peuplé d'informations enregistrées sur l'état des vues de l'interface graphique.

Mais il peut arriver que vous ayez besoin de retenir d'autres informations qui, elles, ne sont pas sauvegardées par défaut. Heureusement, il existe une méthode qui est appelée à chaque fois qu'il y a des chances pour que l'activité soit tuée. Cette méthode s'appelle `protected void onSaveInstanceState(Bundle outState)`.

Un objet de type `Bundle` est l'équivalent d'une table de hachage qui à une chaîne de caractères associe un élément, mais seulement pour certains types précis. Vous pouvez voir dans la documentation tous les types qu'il est possible d'insérer. Par exemple, on peut insérer un entier et le récupérer à l'aide de :

```
```

On ne peut pas mettre n'importe quel objet dans un `Bundle`, uniquement des objets sérialisables. La sérialisation est le procédé qui convertit un objet en un format qui peut être stocké (par exemple dans un fichier ou transmis sur un réseau) et ensuite reconstitué de manière parfaite. Vous faites le rapprochement avec la sérialisation d'une vue en XML, n'est-ce pas ?

En ce qui concerne Android, on n'utilise pas la sérialisation standard de Java, avec l'interface `Java.io.Serializable`, parce que ce processus est trop lent. Or, quand nous essayons de faire communiquer des composants, il faut que l'opération se fasse de manière rapide. C'est pourquoi on utilise un système différent que nous aborderons en détail dans le prochain chapitre.

Cependant, l'implémentation par défaut de `onSaveInstanceState(Bundle)` ne sauvegarde pas toutes les vues, juste celles qui possèdent un identifiant ainsi que la vue qui a le focus, alors n'oubliez pas de faire appel à `super.onSaveInstanceState(Bundle)` pour vous simplifier la vie.

Par la suite, cet objet `Bundle` sera passé à `onCreate(Bundle)`, mais vous pouvez aussi choisir de redéfinir la méthode `onRestoreInstanceState(Bundle)`, qui est appelée après `onStart()` et qui recevra le même objet `Bundle`.

> L'implémentation par défaut de `onRestoreInstanceState(Bundle)` restaure les vues sauvegardées par l'implémentation par défaut de `onSaveInstanceState ()`.

La figure suivante est un schéma qui vous permettra de mieux comprendre tous ces imbroglios.

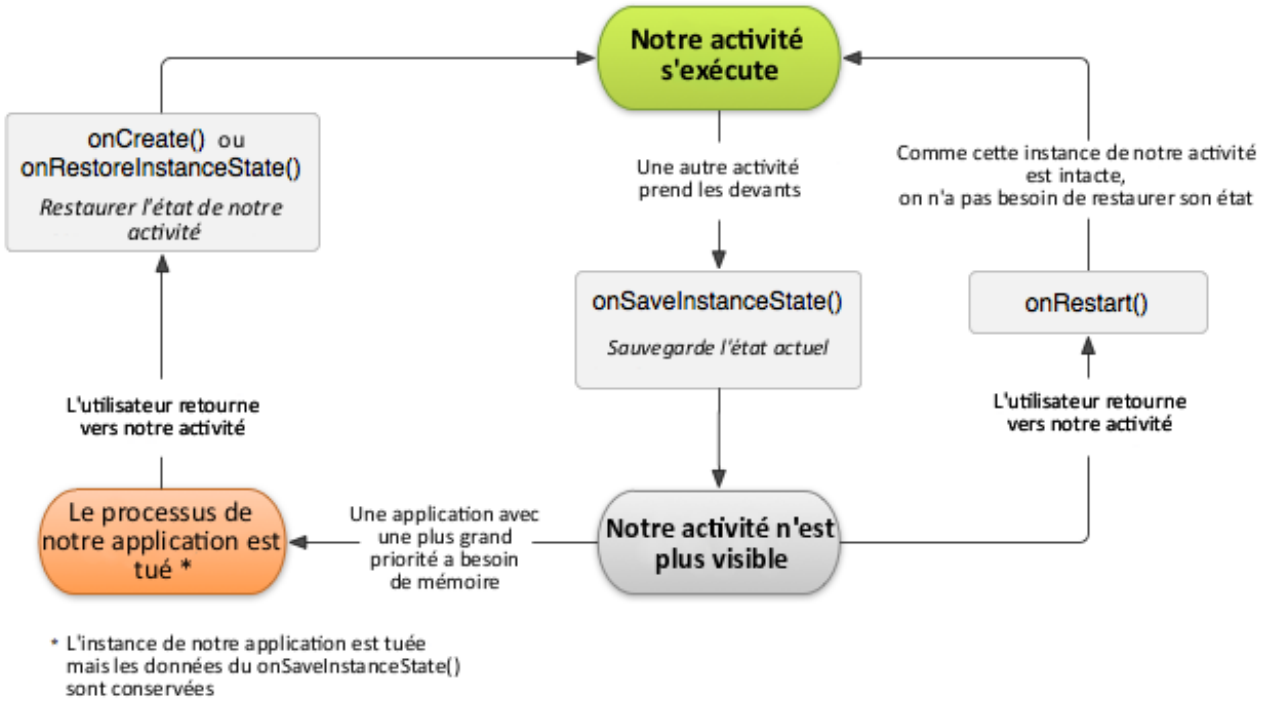

FIGURE 13.3. – Le cycle de sauvegarde de l'état d'une activité

13.5. Gérer le changement de configuration

Il se peut que la configuration de votre utilisateur change pendant qu'il utilise son terminal. Vous allez dire que je suis fou, mais un changement de configuration correspond simplement à ce qui pourrait contribuer à un changement d'interface graphique. Vous vous rappelez les quantificateurs ? Eh bien, si l'un de ces quantificateurs change, alors on dit que la configuration change.

Et ça vous est déjà arrivé, j'en suis sûr. Réfléchissez ! Si l'utilisateur passe de paysage à portrait dans l'un de nos anciens projets, alors il change de configuration et par conséquent d'interface graphique.

Par défaut, dès qu'un changement qui pourrait changer les ressources utilisées par une application se produit, Android détruit tout simplement la ou les activités pour les recréer ensuite.

Heureusement pour vous, Android va retenir les informations des widgets qui possèdent un identifiant. Dans une application très simple, on va créer un layout par défaut :

```

```

Seul un de ces `EditText` possède un identifiant. Ensuite, on fait un layout presque similaire, mais avec un quantificateur pour qu'il ne s'affiche qu'en mode paysage :

```

```

Remarquez bien que l'identifiant de l'`EditText` est passé à l'`EditText` du bas. Ainsi, quand vous lancez votre application, écrivez du texte dans les deux champs, comme à la figure suivante.

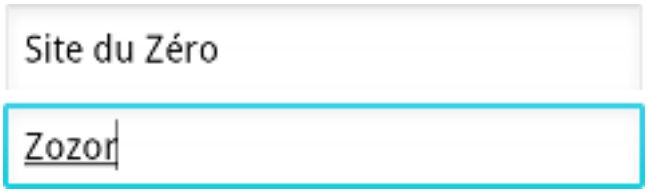

FIGURE 13.4. – Écrivez du texte dans les deux champs

Puis tournez votre terminal (avant qu'un petit malin ne casse son ordinateur en le mettant sur le côté : pour faire pivoter l'émulateur, c'est CTRL + F12 ou F11) et admirez le résultat, identique à la figure suivante.

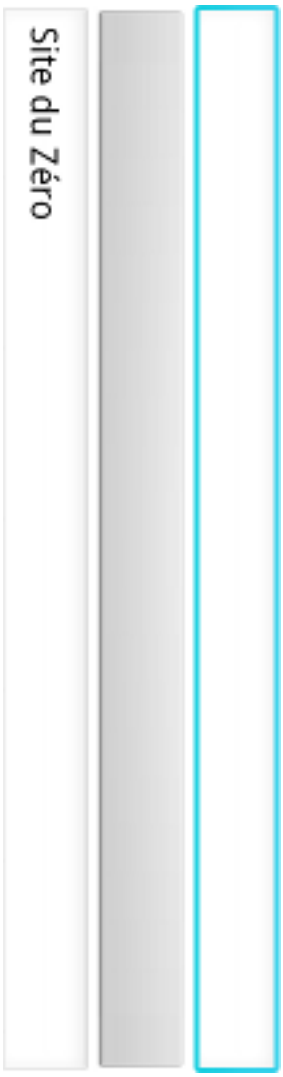

FIGURE 13.5. – Il y a perte d'information !

Vous voyez bien que le widget qui avait un identifiant a conservé son texte, mais pas l'autre ! Cela prouve bien qu'il peut y avoir perte d'information dès qu'un changement de configuration se produit.

Bien entendu, dans le cas des widgets, le problème est vite résolu puisqu'il suffit de leur ajouter un identifiant, mais il existe des informations à retenir en dehors des widgets. Alors, comment gérer ces problèmes-là ? Comme par défaut Android va détruire puis recréer les activités, vous pouvez très bien tout enregistrer dans la méthode `onSaveInstanceState()`, puis tout restaurer dans `onCreate(Bundle)` ou `onRestoreInstanceState()`. Mais il existe un problème ! Vous ne pouvez passer que les objets sérialisables dans un `Bundle`. Alors comment faire ?

Il existe trois façons de faire :

— Utiliser une méthode alternative pour retenir vos objets, qui est spécifique aux changements de configuration.
— Gérer vous-mêmes les changements de configuration, auquel cas Android ne s'en chargera plus. Et comme cette technique est un peu risquée, je ne vais pas vous la présenter.
— Bloquer le changement de ressources.

13.5.0.1. Retenir l'état de l'activité

Donc, le problème avec `Bundle`, c'est qu'il ne peut pas contenir de gros objets et qu'en plus la sérialisation et la désérialisation sont des processus lents, alors que nous souhaiterions que la transition entre deux configurations soit fluide. C'est pourquoi nous allons faire appel à une autre méthode qui est appelée cette fois uniquement en cas de changement de configuration : `public Object onRetainNonConfigurationInstance()`. L'objet retourné peut être de n'importe quel ordre, vous pouvez même retourner directement une instance de l'activité si vous le souhaitez (mais bon, ne le faites pas).

Notez par ailleurs que `onRetainNonConfigurationInstance()` est appelée après `onStop()` mais avant `onDestroy()` et que vous feriez mieux de ne pas conserver des objets qui dépendent de la configuration (par exemple des chaînes de caractères qui changent en fonction de la langue) ou des objets qui sont liés à l'activité (un `Adapter` par exemple).

Ainsi, une des façons de procéder est de créer une classe spécialement dédiée à la détention de ces informations :

```

```

Enfin, il est possible de récupérer cet objet dans le `onCreate(Bundle)` à l'aide de la méthode `public Object getLastNonConfigurationInstance()` :

```

```

13.5.0.2. Empêcher le changement de ressources

De toute façon, il arrive parfois qu'une application n'ait de sens que dans une orientation. Pour lire un livre, il vaut mieux rester toujours en orientation portrait par exemple, de même il est plus agréable de regarder un film en mode paysage. L'idée ici est donc de conserver des fichiers de ressources spécifiques à une configuration, même si celle du terminal change en cours d'utilisation.

Pour ce faire, c'est très simple, il suffit de rajouter dans le nœud des composants concernés les lignes `android:screenOrientation = "portrait"` pour bloquer en mode portrait ou `android:screenOrientation = "landscape"` pour bloquer en mode paysage. Bon, le problème, c'est qu'Android va quand même détruire l'activité pour la recréer si on laisse ça comme ça, c'est pourquoi on va lui dire qu'on gère nous-mêmes les changements d'orientation en ajoutant la ligne `android:configChanges="orientation"` dans les nœuds concernés :

```

```

Voilà, maintenant vous aurez beau tourner le terminal dans tous les sens, l'application restera toujours orientée de la même manière.

— Le fichier Manifest est indispensable à tous les projets Android. C'est lui qui déclarera toute une série d'informations sur votre application.
— Le nœud `<application>` listera les différents composants de votre application ainsi que les services qu'ils offrent.
— Vous pouvez signaler que vous utiliserez des permissions par l'élément `uses-permission` ou que vous en créez par l'élément `permission`.
— Comprendre le cycle de vie d'une activité est essentiel pour construire des activités robustes et ergonomiques.
— Les terminaux peuvent se tenir en mode paysage ou en mode portrait. Vous vous devez de gérer un minimum ce changement de configuration puisqu'au basculement de l'appareil, votre système reconstruira toute votre interface et perdra par conséquent les données que l'utilisateur aurait pu saisir.

14. La communication entre composants

C'est très bien tout ça, mais on ne sait toujours pas comment lancer une activité depuis une autre activité. C'est ce que nous allons voir dans ce chapitre, et même un peu plus. On va apprendre à manipuler un mécanisme puissant qui permet de faire exécuter certaines actions et de faire circuler des messages entre applications ou à l'intérieur d'une même application. Ainsi, chaque application est censée vivre dans un compartiment cloisonné pour ne pas déranger le système quand elle s'exécute et surtout quand elle plante. À l'aide de ces liens qui lient les compartiments, Android devient un vrai puzzle dont chaque pièce apporte une fonctionnalité qui pourrait fournir son aide à une autre pièce, ou au contraire qui aurait besoin de l'aide d'une autre pièce.

Les agents qui sont chargés de ce mécanisme d'échange s'appellent les `intents`. Par exemple, si l'utilisateur clique sur un numéro de téléphone dans votre application, peut-être souhaiteriez-vous que le téléphone appelle le numéro demandé. Avec un intent, vous allez dire à tout le système que vous avez un numéro qu'il faut appeler, et c'est le système qui fera en sorte de trouver les applications qui peuvent le prendre en charge. Ce mécanisme est tellement important qu'Android lui-même l'utilise massivement en interne.

14.1. Aspect technique

Un intent est en fait un objet qui contient plusieurs champs, représentés à la figure suivante.

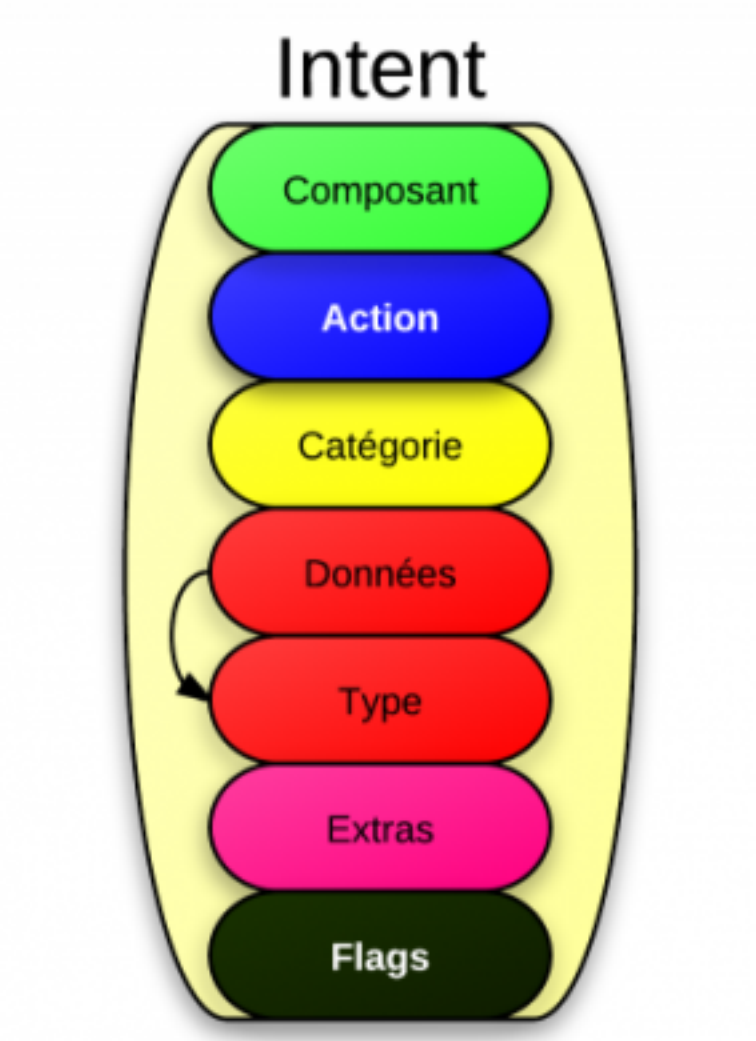

FIGURE 14.1. – Remarquez que le champ « Données » détermine le champ « Type » et que ce n'est pas réciproque

La façon dont sont renseignés ces champs détermine la nature ainsi que les objectifs de l'intent. Ainsi, pour qu'un intent soit dit « explicite », il suffit que son champ composant soit renseigné. Ce champ permet de définir le destinataire de l'intent, celui qui devra le gérer. Ce champ est constitué de deux informations : le *package* où se situe le composant, ainsi que le *nom* du composant. Ainsi, quand l'intent sera exécuté, Android pourra retrouver le composant de destination de manière précise.

À l'opposé des intents explicites se trouvent les intents « implicites ». Dans ce cas de figure, on ne connaît pas de manière précise le destinataire de l'intent, c'est pourquoi on va s'appliquer à renseigner d'autres champs pour laisser Android déterminer qui est capable de réceptionner cet intent. Il faut au moins fournir deux informations essentielles :

— Une action : ce qu'on désire que le destinataire fasse.
— Un ensemble de données : sur quelles données le destinataire doit effectuer son action.

Il existe aussi d'autres informations, pas forcément obligatoires, mais qui ont aussi leur utilité propre le moment venu :

— La catégorie : permet d'apporter des informations supplémentaires sur l'action à exécuter et le type de composant qui devra gérer l'intent.
— Le type : pour indiquer quel est le type des données incluses. Normalement ce type est contenu dans les données, mais en précisant cet attribut vous pouvez désactiver cette vérification automatique et imposer un type particulier.
— Les extras : pour ajouter du contenu à vos intents afin de les faire circuler entre les composants.
— Les flags : permettent de modifier le comportement de l'intent.

14.1.1. Injecter des données dans un intent

14.1.1.1. Types standards

Nous avons vu à l'instant que les intents avaient un champ « extra » qui leur permet de contenir des données à véhiculer entre les applications. Un extra est en fait une clé à laquelle on associe une valeur. Pour insérer un extra, c'est facile, il suffit d'utiliser la méthode `Intent putExtra(String key, X value)` avec `key` la clé de l'extra et `value` la valeur associée. Vous voyez que j'ai mis un `X` pour indiquer le type de la valeur — ce n'est pas syntaxiquement exact, je le sais. Je l'utilise juste pour indiquer qu'on peut y mettre un peu n'importe quel type de base, par exemple `int`, `String` ou `double[]`.

Puis vous pouvez récuperer tous les extras d'un intent à l'aide de la méthode `Bundle getExtras()`, auquel cas vos couples clé-valeurs sont contenus dans le `Bundle`. Vous pouvez encore récupérer un extra précis à l'aide de sa clé et de son type en utilisant la méthode `X get{X}Extra(String key, X defaultValue)`, `X` étant le type de l'extra et `defaultValue` la valeur qui sera retournée si la clé passée ne correspond à aucun extra de l'intent. En revanche, pour les types un peu plus complexes tels que les tableaux, on ne peut préciser de valeur par défaut, par conséquent on devra par exemple utiliser la méthode `float[] getFloatArrayExtra(String key)` pour un tableau de `float`.

En règle générale, la clé de l'extra commence par le package duquel provient l'intent.

Il est possible de rajouter *un unique* `Bundle` en extra avec la méthode `Intent putExtras(Bundle extras)` et *un unique* `Intent` avec la méthode `Intent putExtras(Intent extras)`.

14.1.1.2. Les parcelables

Cependant, `Bundle` ne peut pas prendre tous les objets, comme je vous l'ai expliqué précédemment, il faut qu'ils soient sérialisables. Or, dans le cas d'Android, on considère qu'un objet est sérialisable à partir du moment où il implémente correctement l'interface `Parcelable` . Si on devait entrer dans les détails, sachez qu'un `Parcelable` est un objet qui sera transmis à un `Parcel` , et que l'objectif des `Parcel` est de transmettre des messages entre différents processus du système.

Pour implémenter l'interface `Parcelable`, il faut redéfinir deux méthodes :

— `int describeContents()`, qui permet de définir si vous avez des paramètres spéciaux dans votre `Parcelable`. En ce mois de juillet 2012 (à l'heure où j'écris ces lignes), les seuls objets spéciaux à considérer sont les `FileDescriptor` . Ainsi, si votre objet ne contient pas d'objet de type `FileDescriptor`, vous pouvez renvoyer 0, sinon renvoyez `Parcelable.CONTENT_FILE_DESCRIPTOR`.
— `void writeToParcel(Parcel dest, int flags)`, avec `dest` le `Parcel` dans lequel nous allons insérer les attributs de notre objet et `flags` un entier qui vaut la plupart du temps 0. C'est dans cette classe que nous allons écrire dans le `Parcel` qui transmettra le message.

Les attributs sont à insérer dans le `Parcel` dans l'ordre dans lequel ils sont déclarés dans la classe !

Si on prend l'exemple simple d'un contact dans un répertoire téléphonique :

Tous nos attributs sont désormais dans le `Parcel`, on peut transmettre notre objet.

C'est presque fini, cependant, il nous faut encore ajouter un champ statique de type `Parcelable.Creator` et qui s'appellera impérativement « CREATOR », sinon nous serions incapables de reconstruire un objet qui est passé par un `Parcel` :

Enfin, comme n'importe quel autre objet, on peut l'ajouter dans un intent avec `putExtra` et on peut le récupérer avec `getParcelableExtra`.

14.2. Les intents explicites

Créer un intent explicite est très simple puisqu'il suffit de donner un `Context` qui appartienne au package où se trouve la classe de destination :

Par exemple, si la classe de destination appartient au package du `Context` actuel :

À noter qu'on aurait aussi pu utiliser la méthode `Intent setClass(Context packageContext, Class<?> cls)` avec `packageContext` un `Context` qui appartient au même package que le composant de destination et `cls` le nom de la classe qui héberge cette activité.

Il existe ensuite deux façons de lancer l'intent, selon qu'on veuille que le composant de destination nous renvoie une réponse ou pas.

14.2.1. Sans retour

Si vous ne vous attendez pas à ce que la nouvelle activité vous renvoie un résultat, alors vous pouvez l'appeler très naturellement avec `void startActivity (Intent intent)` dans votre activité. La nouvelle activité sera indépendante de l'actuelle. Elle entreprendra un cycle d'activité normal, c'est-à-dire en commençant par un `onCreate`.

Voici un exemple tout simple : dans une première activité, vous allez mettre un bouton et vous allez faire en sorte qu'appuyer sur ce bouton lance une seconde activité :

```

```

La seconde activité ne fera rien de particulier, si ce n'est afficher un layout différent :

```

```

Enfin, n'oubliez pas de préciser dans le Manifest que vous avez désormais deux activités au sein de votre application :

```

```

Ainsi, dès qu'on clique sur le bouton de la première activité, on passe directement à la seconde activité, comme le montre la figure suivante.

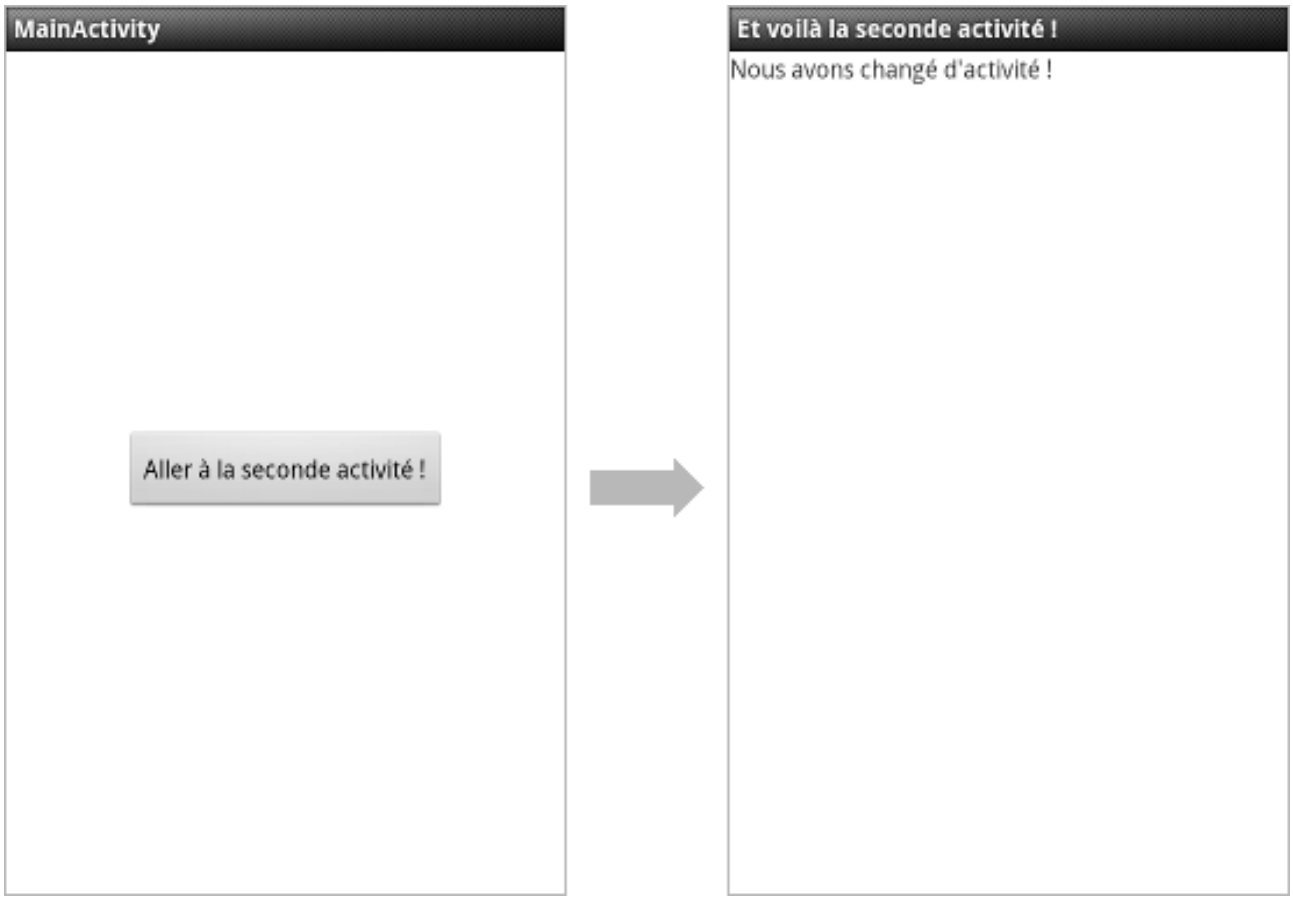

FIGURE 14.2. – En cliquant sur le bouton de la première activité, on passe à la seconde

14.2.2. Avec retour

Cette fois, on veut qu'au retour de l'activité qui vient d'être appelée cette dernière nous renvoie un petit *feedback*. Pour cela, on utilisera la méthode `void startActivityForResult(Intent intent, int requestCode)`, avec `requestCode` un code passé qui permet d'identifier de manière unique un intent.

Ce code doit être supérieur ou égal à 0, sinon Android considérera que vous n'avez pas demandé de résultat.

Quand l'activité appelée s'arrêtera, la première méthode de *callback* appelée dans l'activité précédente sera `void onActivityResult(int requestCode, int resultCode, Intent data)`. On retrouve `requestCode`, qui sera le même code que celui passé dans le `startActivityForResult` et qui permet de repérer quel intent a provoqué l'appel de l'activité dont le cycle vient de s'interrompre. `resultCode` est quant à lui un code renvoyé par l'activité qui indique comment elle s'est terminée (typiquement `Activity.RESULT_OK` si l'activité s'est terminée normalement, ou `Activity.RESULT_CANCELED` s'il y a eu un problème ou qu'aucun code de retour n'a été précisé). Enfin, `intent` est un intent qui contient éventuellement des données.

Par défaut, le code renvoyé par une activité est `Activity.RESULT_CANCELED` de façon que, si l'utilisateur utilise le bouton Retour avant que l'activité ait fini de s'exécuter, vous puissiez savoir que le résultat fourni ne sera pas adapté à vos besoins.

Dans la seconde activité, vous pouvez définir un résultat avec la méthode `void setResult(int resultCode, Intent data)`, ces paramètres étant identiques à ceux décrits ci-dessus.

Ainsi, l'attribut `requestCode` de `void startActivityForResult(Intent intent, int requestCode)` sera similaire au `requestCode` que nous fournira la méthode de *callback* `void onActivityResult(int requestCode, int resultCode, Intent data)`, de manière à pouvoir identifier quel intent est à l'origine de ce retour.

Le code de ce nouvel exemple sera presque similaire à celui de l'exemple précédent, sauf que cette fois la seconde activité proposera à l'utilisateur de cliquer sur deux boutons. Cliquer sur un de ces boutons retournera à l'activité précédente en lui indiquant lequel des deux boutons a été pressé. Ainsi, `MainActivity` ressemble désormais à :

```

```

Alors que la seconde activité devient :

```

```

Et voilà, dès que vous cliquez sur un des boutons, la première activité va lancer un `Toast` qui affichera quel bouton a été pressé, comme le montre la figure suivante.

FIGURE 14.3. – Un Toast affiche quel bouton a été pressé

14.3. Les intents implicites

Ici, on fera en sorte d'envoyer une requête à un destinataire, sans savoir qui il est, et d'ailleurs on s'en fiche tant que le travail qu'on lui demande de faire est effectué. Ainsi, les applications destinataires sont soit fournies par Android, soit par d'autres applications téléchargées sur le Play Store par exemple.

14.3.1. Les données

14.3.1.1. L'URI

La première chose qu'on va étudier, c'est les données, parce qu'elles sont organisées selon une certaine syntaxe qu'il vous faut connaître. En fait, elles sont formatées à l'aide des URI. Un URI est une chaîne de caractères qui permet d'identifier un endroit. Par exemple sur internet, ou dans le cas d'Android sur le périphérique ou une ressource. Afin d'étudier les URI, on va faire l'analogie avec les adresses URL qui nous permettent d'accéder à des sites internet. En effet, un peu à la manière d'un serveur, nos fournisseurs de contenu vont répondre en fonction de l'URI fournie.

De plus, la forme générale d'une URI rappelle fortement les URL. Prenons l'exemple du Site du Zéro avec une URL inventée : `http://www.siteduzero.com/forum/android/aide.html`. On identifie plusieurs parties :

— `http://`
— `www.siteduzero.com`
— `/forum/android/aide.html`

Les URI se comportent d'une manière un peu similaire. La syntaxe d'un URI peut être analysée de la manière suivante (les parties entre accolades {} sont optionnelles) :

```

```

— Le `schéma` décrit quelle est la nature de l'information. S'il s'agit d'un numéro de téléphone, alors le schéma sera `tel`, s'il s'agit d'un site internet, alors le schéma sera `http`, etc.
— L'`information` est la donnée en tant que telle. Cette information respecte elle aussi une syntaxe, mais qui dépend du schéma cette fois-ci. Ainsi, pour un numéro de téléphone, vous pouvez vous contenter d'insérer le numéro `tel:0606060606`, mais pour des coordonnées GPS il faudra séparer la latitude de la longitude à l'aide d'une virgule `geo:123.456789,-12.345678`. Pour un site internet, il s'agit d'un chemin hiérarchique.
— La `requête` permet de fournir une précision par rapport à l'information.
— Le `fragment` permet enfin d'accéder à une sous-partie de l'information.

Pour créer un objet URI, c'est simple, il suffit d'utiliser la méthode statique `Uri Uri.parse(String uri)`. Par exemple, pour envoyer un SMS à une personne, j'utiliserai l'URI :

```

```

Mais je peux aussi indiquer plusieurs destinataires et un corps pour ce message :

```

```

Comme vous pouvez le voir, le contenu de la chaîne doit être encodé, sinon vous rencontrerez des problèmes.

14.3.1.2. Type MIME

Le MIME est un identifiant pour les formats de fichier. Par exemple, il existe un type MIME `text`. Si une donnée est accompagnée du type MIME `text`, alors les données sont du texte. On trouve aussi `audio` et `video` par exemple. Il est ensuite possible de préciser un sous-type afin d'affiner les informations sur les données, par exemple `audio/mp3` et `audio/wav` sont deux types MIME qui indiquent que les données sont sonores, mais aussi de quelle manière elles sont encodées.

Les types MIME que nous venons de voir son standards, c'est-à-dire qu'il y a une organisation qui les a reconnus comme étant légitimes. Mais si vous vouliez créer vos propres types MIME ? Vous n'allez pas demander à l'organisation de les valider, ils ne seront pas d'accord avec vous. C'est

pourquoi il existe une petite syntaxe à respecter pour les types personnalisés : `vnd.votre_package.le_type`, ce qui peut donner par exemple `vnd.sdz.chapitreTrois.contact_telephonique`.

Pour être tout à fait exact, sous Android vous ne pourrez jamais que préciser des sous-types, jamais des types.

Pour les intents, ce type peut être décrit de manière implicite dans l'URI (on voit bien par exemple que `sms:0606060606` décrit un numéro de téléphone, il n'est pas nécessaire de le préciser), mais il faudra par moments le décrire de manière explicite. On peut le faire dans le champ `type` d'un intent. Vous trouverez une liste non exhaustive des types MIME sur cette page Wikipédia .

Préciser un type est surtout indispensable quand on doit manipuler des ensembles de données, comme par exemple quand on veut supprimer une ou plusieurs entrées dans le répertoire, car dans ce cas précis il s'agira d'un pointeur vers ces données. Avec Android, il existe deux manières de manipuler ces ensembles de données, les curseurs (*cursor*) et les fournisseurs de contenus (*content provider*). Ces deux techniques seront étudiées plus tard, par conséquent nous allons nous cantonner aux données simples pour l'instant.

14.3.2. L'action

Une action est une constante qui se trouve dans la classe `Intent` et qui commence toujours par « ACTION » *suivi d'un verbe (en anglais, bien sûr) de façon à bien faire comprendre qu'il s'agit d'une action. Si vous voulez _voir* quelque chose, on va utiliser l'action `ACTION_VIEW`. Par exemple, si vous utilisez `ACTION_VIEW` sur un numéro de téléphone, alors le numéro de téléphone s'affichera dans le composeur de numéros de téléphone.

Vous pouvez aussi créer vos propres actions. Pour cela, il vaux mieux respecter une syntaxe, qui est de commencer par votre package suivi de `.intent.action.NOM_DE_L_ACTION` :

```
```

Voici quelques actions natives parmi les plus usitées :

Intitulé	Action	Entrée attendue	Sortie attendue
`ACTION_MAIN`	**Pour indiquer qu'il s'agit du point d'entrée dans l'application**	/	/
`ACTION_DIAL`	Pour ouvrir le composeur de numéros téléphoniques	Un numéro de téléphone semble une bonne idée :-p	/
`ACTION_DELETE*`	Supprimer des données	Un URI vers les données à supprimer	/

`ACTION_EDIT*`	Ouvrir un éditeur adapté pour modifier les données fournies	Un URI vers les données à éditer	/
`ACTION_INSERT*`	Insérer des données	L'URI du répertoire où insérer les données	L'URI des nouvelles données créées
`ACTION_PICK*`	Sélectionner un élément dans un ensemble de données	L'URI qui contient un répertoire de données à partir duquel l'élément sera sélectionné	L'URI de l'élément qui a été sélectionné
`ACTION_SEARCH`	Effectuer une recherche	Le texte à rechercher	/
`ACTION_SENDTO`	Envoyer un message à quelqu'un	La personne à qui envoyer le message	/
`ACTION_VIEW`	Permet de visionner une donnée	Un peu tout. Une adresse e-mail sera visionnée dans l'application pour les e-mails, un numéro de téléphone dans le composeur, une adresse internet dans le navigateur, etc.	/
ACTION_WEB_SEARCH	Effectuer une recherche sur internet	S'il s'agit d'un texte qui commence par « http », le site s'affichera directement, sinon c'est une recherche dans Google qui se fera	/

Les actions suivies d'une astérisque sont celles que vous ne pourrez pas utiliser tant que nous n'aurons pas vu les `Content Provider`.

Pour créer un intent qui va ouvrir le composeur téléphonique avec le numéro de téléphone 0606060606, j'adapte mon code précédent en remplaçant le code du bouton par :

```
```

Ce qui donne, une fois que j'appuie dessus, la figure suivante.

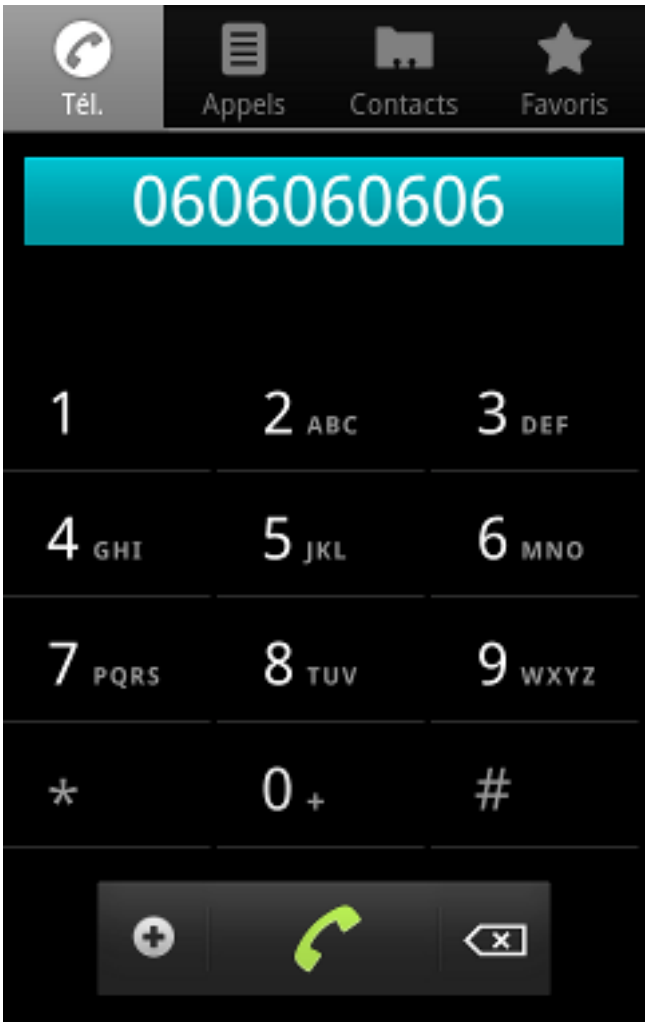

FIGURE 14.4. – Le composeur téléphonique est lancé avec le numéro souhaité

14.4. La résolution des intents

Quand on lance un `ACTION_VIEW` avec une adresse internet, c'est le navigateur qui se lance, et quand on lance un `ACTION_VIEW` avec un numéro de téléphone, c'est le composeur de numéros qui se lance. Alors, comment Android détermine qui doit répondre à un intent donné ?

Ce que va faire Android, c'est qu'il va comparer l'intent à des filtres que nous allons déclarer dans le Manifest et qui signalent que les composants de nos applications peuvent gérer certains intents. Ces filtres sont les nœuds `<intent-filter>`, nous les avons déjà rencontrés et ignorés par le passé. Un composant d'une application doit avoir autant de filtres que de capacités de traitement. S'il peut gérer deux types d'intent, il doit avoir deux filtres.

Le test de conformité entre un intent et un filtre se fait sur trois critères.

14.4.1. L'action

Permet de filtrer en fonction du champ `Action` d'un intent. Il peut y en avoir un ou plus par filtre. Si vous n'en mettez pas, tous vos intents seront recalés. Un intent sera accepté si ce qui

se trouve dans son champ `Action` est identique à au moins une des actions du filtre. Et si un intent ne précise pas d'action, alors il sera automatiquement accepté pour ce test.

C'est pour cette raison que les intents explicites sont toujours acceptés, ils n'ont pas de champ `Action`, par conséquent ils passent le test, même si le filtre ne précise aucune action.

Cette activité ne pourra intercepter que les intents qui ont dans leur champ action `ACTION_VIEW` et/ou `ACTION_SENDTO`, car *toutes* ses actions sont acceptées par le filtre. Si un intent a pour action `ACTION_VIEW` et `ACTION_SEARCH`, alors il sera recalé, car une de ses actions n'est pas acceptée par le filtre.

14.4.2. La catégorie

Cette fois, il n'est pas indispensable d'avoir une indication de catégorie pour un intent, mais, s'il y en a une ou plusieurs, alors pour passer ce test il faut que *toutes* les catégories de l'intent correspondent à des catégories du filtre. Pour les matheux, on dit qu'il s'agit d'une application « injective » mais pas « surjective ».

On pourrait se dire que par conséquent, si un intent n'a pas de catégorie, alors il passe automatiquement ce test, mais dès qu'un intent est utilisé avec la méthode `startActivity()`, alors on lui ajoute la catégorie `CATEGORY_DEFAULT`. Donc, si vous voulez que votre composant accepte les intents implicites, vous devez rajouter cette catégorie à votre filtre.

Pour les actions et les catégories, la syntaxe est différente entre le Java et le XML. Par exemple, pour l'action `ACTION_VIEW` en Java, on utilisera `android.intent.action.VIEW` et pour la categorie `CATEGORY_DEFAULT` on utilisera `android.intent.category.DEFAULT`. De plus, quand vous créez vos propres actions ou catégories, le mieux est de les préfixer avec le nom de votre package afin de vous assurer qu'elles restent uniques. Par exemple, pour l'action `DESEMBROUILLER`, on pourrait utiliser `sdz.chapitreQuatre.action.DESEMBROUILLER`.

Il faut ici que l'intent ait pour action `ACTION_VIEW` et/ou `ACTION_SEARCH`. En ce qui concerne les catégories, il doit avoir `CATEGORY_DEFAULT` *et* `CATEGORY_DESEMBROUILLEUR`.

Voici les principales catégories par défaut fournies par Android :

Catégorie	Description

`CATEGORY_DEFAULT`	**Indique qu'il faut effectuer le traitement par défaut sur les données correspondantes. Concrètement, on l'utilise pour déclarer qu'on accepte que ce composant soit utilisé par des intents implicites.**
`CATEGORY_BROWSABLE`	Utilisé pour indiquer qu'une activité peut être appelée sans risque depuis un navigateur web. Ainsi, si un utilisateur clique sur un lien dans votre application, vous promettez que rien de dangereux ne se passera à la suite de l'activation de cet intent.
`CATEGORY_TAB`	Utilisé pour les activités qu'on retrouve dans des onglets.
`CATEGORY_ALTERNATIVE`	Permet de définir une activité comme un traitement alternatif dans le visionnage d'éléments. C'est par exemple intéressant dans les menus, si vous souhaitez proposer à votre utilisateur de regarder telles données de la manière proposée par votre application ou d'une manière que propose une autre application.
`CATEGORY_SELECTED_ALTERNATIVE`	Comme ci-dessus, mais pour des éléments qui ont été sélectionnés, pas seulement pour les voir.
`CATEGORY_LAUNCHER`	Indique que c'est ce composant qui doit s'afficher dans le lanceur d'applications.
`CATEGORY_HOME`	Permet d'indiquer que c'est cette activité qui doit se trouver sur l'écran d'accueil d'Android.
`CATEGORY_PREFERENCE`	Utilisé pour identifier les `PreferenceActivity` (dont nous parlerons au chapitre suivant).

14.4.3. Les données

Il est possible de préciser plusieurs informations sur les données que cette activité peut traiter. Principalement, on peut préciser le schéma qu'on veut avec `android:scheme`, on peut aussi préciser le type MIME avec `android:mimeType`. Par exemple, si notre application traite des fichiers textes qui proviennent d'internet, on aura besoin du type « texte » et du schéma « internet », ce qui donne :

```
```

?

Et il se passe quoi en interne une fois qu'on a lancé un intent ?

Eh bien, il existe plusieurs cas de figure :

— Soit votre recherche est infructueuse et vous avez 0 résultat, auquel cas c'est grave et une `ActivityNotFoundException` sera lancée. Il vous faut donc penser à gérer ce type d'erreurs.
— Si on n'a qu'un résultat, comme dans le cas des intents explicites, alors ce résultat va directement se lancer.
— En revanche, si on a plusieurs réponses possibles, alors le système va demander à l'utilisateur de choisir à l'aide d'une boîte de dialogue. Si l'utilisateur choisit une action par défaut pour un intent, alors à chaque fois que le même intent sera émis ce sera toujours le même composant qui sera sélectionné. D'ailleurs, il peut arriver que ce soit une mauvaise chose parce qu'un même intent ne signifie pas toujours une même intention (ironiquement). Il se peut qu'avec ACTION*SEND, on cherche un jour à envoyer un SMS et un autre jour à envoyer un e-mail, c'est pourquoi il est possible de forcer la main à Android et à obliger l'utilisateur à choisir parmi plusieurs éventualités à l'aide de* `Intent createChooser(Intent target, CharSequence titre)`*. On peut ainsi insérer l'_intent* à traiter et le *titre* de la boîte de dialogue qui permettra à l'utilisateur de choisir une application.

Dans tous les cas, vous pouvez vérifier si un composant va réagir à un intent de manière programmatique à l'aide du `Package Manager` . Le `Package Manager` est un objet qui vous permet d'obtenir des informations sur les packages qui sont installés sur l'appareil. On y fait appel avec la méthode `PackageManager getPackageManager()` dans n'importe quel composant. Puis on demande à l'intent le nom de l'activité qui va pouvoir le gérer à l'aide de la méthode `ComponentName resolveActivity (PackageManager pm)` :

```
```

14.5. Pour aller plus loin : navigation entre des activités

Une application possède en général plusieurs activités. Chaque activité est dédiée à un ensemble cohérent d'actions, mais toujours centrées vers un même objectif. Pour une application qui lit des musiques, il y a une activité pour choisir la musique à écouter, une autre qui présente les contrôles sur les musiques, encore une autre pour paramétrer l'application, etc.

Je vous avais présenté au tout début du tutoriel la pile des activités. En effet, comme on ne peut avoir qu'une activité visible à la fois, les activités étaient présentées dans une pile où il était possible d'ajouter ou d'enlever un élément au sommet afin de changer l'activité consultée actuellement. C'est bien sûr toujours vrai, à un détail près. Il existe en fait une pile par tâche. On pourrait dire qu'une tâche est une application, mais aussi les activités qui seront lancées par cette application et qui sont extérieures à l'application. Ainsi que les activités qui seront lancées par ces activités extérieures, etc.

Au démarrage de l'application, une nouvelle tâche est créée et l'activité principale occupe la racine de la pile.

Au lancement d'une nouvelle activité, cette dernière est ajoutée au sommet de la pile et acquiert ainsi le focus. L'activité précédente est arrêtée, mais l'état de son interface graphique est conservé. Quand l'utilisateur appuie sur le bouton Retour, l'activité actuelle est éjectée de la pile (elle est

donc détruite) et l'activité précédente reprend son déroulement normal (avec restauration des éléments de l'interface graphique). S'il n'y a pas d'activité précédente, alors la tâche est tout simplement détruite.

Dans une pile, on ne manipule jamais que le sommet. Ainsi, si l'utilisateur appuie sur un bouton de l'activité 1 pour aller à l'activité 2, puis appuie sur un bouton de l'activité 2 pour aller dans l'activité 1, alors une nouvelle instance de l'activité 1 sera créée, comme le montre la figure suivante.

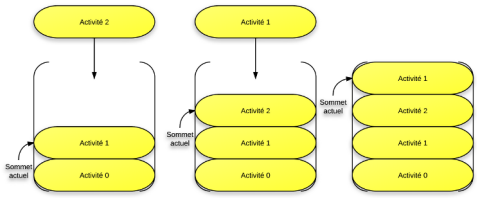

FIGURE 14.5. – On passe de l'activité 1 à l'activité 2, puis de l'activité 2 à l'activité 1, ce qui fait qu'on a deux différentes instances de l'activité 1 !

Pour changer ce comportement, il est possible de manipuler l'**affinité** d'une activité. Cette affinité est un attribut qui indique avec quelle tâche elle préfère travailler. Toutes les activités qui ont une affinité avec une même tâche se lanceront dans cette tâche-là.

Elle permet surtout de déterminer à quelle tâche une activité sera apparentée ainsi que la tâche qui va accueillir l'activité quand elle est lancée avec le flag `FLAG_ACTIVITY_NEW_TASK`. Par défaut, toutes les activités d'une même application ont la même affinité. En effet, si vous ne précisez pas d'affinité, alors cet attribut prendra la valeur du package de l'application.

Ce comportement est celui qui est préférable la plupart du temps. Cependant, il peut arriver que vous ayez besoin d'agir autrement, auquel cas il y a deux façons de faire.

14.5.1. Modifier l'activité dans le Manifest

Il existe six attributs que nous n'avons pas encore vus et qui permettent de changer la façon dont Android réagit à la navigation.

14.5.1.1. android :taskAffinity

Cet attribut permet de préciser avec quelle tâche cette activité possède une affinité. Exemple :

14.5.1.2. android :allowTaskReparenting

Est-ce que l'activité peut se déconnecter d'une tâche dans laquelle elle a commencé à travailler pour aller vers une autre tâche avec laquelle elle a une affinité ?

Par exemple, dans le cas d'une application pour lire les SMS, si le SMS contient un lien, alors cliquer dessus lancera une activité qui permettra d'afficher la page web désignée par le lien. Si on appuie sur le bouton Retour, on revient à la lecture du SMS. En revanche, avec cet attribut, l'activité lancée sera liée à la tâche du navigateur et non plus du client SMS.

La valeur par défaut est `false`.

14.5.1.3. android :launchMode

Définit comment l'application devra être lancée dans une tâche. Il existe deux modes : soit l'activité peut être instanciée plusieurs fois dans la même tâche, soit elle est toujours présente de manière unique.

Dans le premier mode, il existe deux valeurs possibles :

- `standard` est le mode par défaut, dès qu'on lance une activité une nouvelle instance est créée dans la tâche. Les différentes instances peuvent aussi appartenir à plusieurs tâches.
- Avec `singleTop`, si une instance de l'activité existe déjà au sommet de la tâche actuelle, alors le système redirigera l'intent vers cette instance au lieu de créer une nouvelle instance. Le retour dans l'activité se fera à travers la méthode de *callback* `void onNewIntent(Intent intent)`.

Le second mode n'est pas recommandé et doit être utilisé uniquement dans des cas précis. Surtout, on ne l'utilise que si l'activité est celle de lancement de l'application. Il peut prendre deux valeurs :

- Avec `singleTask`, le système crée l'activité à la racine d'une nouvelle tâche. Cependant, si une instance de l'activité existe déjà, alors on ouvrira plutôt cette instance-là.
- Enfin avec `singleInstance`, à chaque fois on crée une nouvelle tâche dont l'activité sera la racine.

14.5.1.4. android :clearTaskOnLaunch

Est-ce que toutes les activités doivent être enlevées de la tâche — à l'exception de la racine — quand on relance la tâche depuis l'écran de démarrage ? Ainsi, dès que l'utilisateur relance l'application, il retournera à l'activité d'accueil, sinon il retournera dans la dernière activité qu'il consultait.

La valeur par défaut est `false`.

14.5.1.5. android :alwaysRetainTaskState

Est-ce que l'état de la tâche dans laquelle se trouve l'activité — et dont elle est la racine — doit être maintenu par le système ?

Typiquement, une tâche est détruite si elle n'est pas active et que l'utilisateur ne la consulte pas pendant un certain temps. Cependant, dans certains cas, comme dans le cas d'un navigateur web avec des onglets, l'utilisateur sera bien content de récupérer les onglets qui étaient ouverts.

La valeur par défaut est `false`.

14.5.1.6. android :finishOnTaskLaunch

Est-ce que, s'il existe déjà une instance de cette activité, il faut la fermer dès qu'une nouvelle instance est demandée ?

La valeur par défaut est `false`.

14.5.2. Avec les intents

Il est aussi possible de modifier l'association par défaut d'une activité à une tâche à l'aide des flags contenus dans les intents. On peut rajouter un flag à un intent avec la méthode `Intent addFlags(int flags)`.

Il existe trois flags principaux :

- `FLAG_ACTIVITY_NEW_TASK` permet de lancer l'activité dans une nouvelle tâche, sauf si l'activité existe déjà dans une tâche. C'est l'équivalent du mode `singleTask`.
- `FLAG_ACTIVITY_SINGLE_TOP` est un équivalent de `singleTop`. On lancera l'activité dans une nouvelle tâche, quelques soient les circonstances.
- `FLAG_ACTIVITY_CLEAR_TOP` permet de faire en sorte que, si l'activité est déjà lancée dans la tâche actuelle, alors au lieu de lancer une nouvelle instance de cette activité toutes les autres activités qui se trouvent au-dessus d'elle seront fermées et l'intent sera délivré à l'activité (souvent utilisé avec `FLAG_ACTIVITY_NEW_TASK`). Quand on utilise ces deux flags ensemble, ils permettent de localiser une activité qui existait déjà dans une autre tâche et de la mettre dans une position où elle pourra répondre à l'intent.

14.6. Pour aller plus loin : diffuser des intents

On a vu avec les intents comment dire « Je veux que vous traitiez cela, alors que quelqu'un le fasse pour moi s'il vous plaît ». Ici on va voir comment dire « Cet évènement vient de se dérouler, je préviens juste, si cela intéresse quelqu'un ». C'est donc la différence entre « Je viens de recevoir un SMS, je cherche un composant qui pourra permettre à l'utilisateur de lui répondre » et « Je viens de recevoir un SMS, ça intéresse une application de le gérer ? ». Il s'agit ici uniquement de notifications, pas de demandes. Concrètement, le mécanisme normal des intents est visible pour l'utilisateur, alors que celui que nous allons étudier est totalement transparent pour lui.

Nous utiliserons toujours des intents, sauf qu'ils seront anonymes et diffusés à tout le système. Ce type d'intent est très utilisé au niveau du système pour transmettre des informations, comme par exemple l'état de la batterie ou du réseau. Ces intents particuliers s'appellent des *broadcast intents*. On utilise encore une fois un système de filtrage pour déterminer qui peut recevoir l'intent, mais c'est la façon dont nous allons recevoir les messages qui est un peu spéciale.

La création des broadcast intents est similaire à celle des intents classiques, sauf que vous allez les envoyer avec la méthode `void sendBroadcast(Intent intent)`. De cette manière, l'intent ne sera reçu que par les *broadcast receivers*, qui sont des classes qui dérivent de la classe `BroadcastReceiver` . De plus, quand vous allez déclarer ce composant dans votre Manifest, il faudra que vous annonciez qu'il s'agit d'un broadcast receiver :

```

```

Il vous faudra alors redéfinir la méthode de *callback* `void onReceive (Context context, Intent intent)` qui est lancée dès qu'on reçoit un broadcast intent. C'est dans cette classe qu'on gérera le message reçu.

Par exemple, si j'ai un intent qui transmet à tout le système le nom de l'utilisateur :

```

```

Un broascast receiver déclaré de cette manière sera disponible tout le temps, même quand l'application n'est pas lancée, mais ne sera viable que pendant la durée d'exécution de sa méthode `onReceive`. Ainsi, ne vous attendez pas à retrouver votre receiver si vous lancez un thread, une boîte de dialogue ou un autre composant d'une application à partir de lui.

De plus, il ne s'exécutera pas en parallèle de votre application, mais bien de manière séquentielle (dans le même thread, donc), ce qui signifie que, si vous effectuez de gros calculs qui prennent du temps, les performances de votre application pourraient s'en trouver affectées.

Mais il est aussi possible de déclarer un broadcast receiver de manière dynamique, directement dans le code. Cette technique est surtout utilisée pour gérer les évènements de l'interface graphique.

Pour procéder, vous devrez créer une classe qui dérive de `BroadcastReceiver`, mais sans l'enregistrer dans le Manifest. Ensuite, vous pouvez lui rajouter des lois de filtrage avec la classe `IntentFilter`, puis vous pouvez l'enregistrer dans l'activité voulue avec la méthode `Intent registerReceiver(BroadcastReceiver receiver, IntentFilter filter)` et surtout, quand vous n'en n'aurez plus besoin, il faudra la désactiver avec `void unregisterReceiver(BroadcastReceiver receiver).`

Ainsi, si on veut recevoir nos broadcast intents pour dire coucou à l'utilisateur, mais uniquement quand l'application se lance et qu'elle n'est pas en pause, on fait :

```

```

De plus, il existe quelques messages diffusés par le système de manière native et que vous pouvez écouter, comme par exemple `ACTION_CAMERA_BUTTON` qui est lancé dès que l'utilisateur appuie sur le bouton de l'appareil photo.

14.6.0.1. Sécurité

N'importe quelle application peut envoyer des broadcast intents à votre receiver, ce qui est une faiblesse au niveau sécurité. Vous pouvez aussi faire en sorte que votre receiver déclaré dans le Manifest ne soit accessible qu'à l'intérieur de votre application en lui ajoutant l'attribut `android:exported="false"` :

```
```

Notez que cet attribut est disponible pour tous les composants.

De plus, quand vous envoyez un broadcast intent, toutes les applications peuvent le recevoir. Afin de déterminer qui peut recevoir un broadcast intent, il suffit de lui ajouter une permission à l'aide de la méthode `void sendBroadcast (Intent intent, String receiverPermission)`, avec `receiverPermission` une permission que vous aurez déterminée. Ainsi, seuls les receivers qui déclarent cette permission pourront recevoir ces broadcast intents :

```
```

Puis dans le Manifest, il suffit de rajouter :

```
```

— Les intents sont des objets permettant d'envoyer des messages entre vos activités, voire entre vos applications. Ils peuvent, selon vos préférences, contenir un certain nombre d'informations qu'il sera possible d'exploiter dans une autre activité.
— En général, les données contenues dans un intent sont assez limitées mais il est possible de partager une classe entière si vous étendez la classe `Parcelable` et que vous implémentez toutes les méthodes nécessaires à son fonctionnement.
— Les intents explicites sont destinés à se rendre à une activité très précise. Vous pourriez également définir que vous attendez un retour suite à cet appel via la méthode `void startActivityForResult(Intent intent, int requestCode)`.
— Les intents implicites sont destinés à demander à une activité, sans que l'on sache laquelle, de traiter votre message en désignant le type d'action souhaité et les données à traiter.
— Définir un nœud `<intent-filter>` dans le nœud d'une `<activity>` de votre fichier Manifest vous permettra de filtrer vos activités en fonction du champ d'action de vos intents.
— Nous avons vu qu'Android gère nos activités *via* une pile LIFO. Pour changer ce comportement, il est possible de manipuler l'affinité d'une activité. Cette affinité est un attribut qui indique avec quelle tâche elle préfère travailler.

— Les broadcast intents diffusent des intents à travers tout le système pour transmettre des informations de manière publique à qui veut. Cela se met en place grâce à un nœud `<receiver>` filtré par un nœud `<intent-filter>`.

15. Le stockage de données

La plupart de nos applications auront besoin de stocker des données à un moment ou à un autre. La couleur préférée de l'utilisateur, sa configuration réseau ou encore des fichiers téléchargés sur internet. En fonction de ce que vous souhaitez faire et de ce que vous souhaitez enregistrer, Android vous fournit plusieurs méthodes pour sauvegarder des informations. Il existe deux solutions qui permettent d'enregistrer des données de manière rapide et flexible, si on exclut les bases de données :

— Nous aborderons tout d'abord les préférences partagées, qui permettent d'associer à un identifiant une valeur. Le couple ainsi créé permet de retenir les différentes options que l'utilisateur souhaiterait conserver ou l'état de l'interface graphique. Ces valeurs pourront être partagées entre plusieurs composants. Encore mieux, Android propose un ensemble d'outils qui permettront de faciliter grandement le travail et d'unifier les interfaces graphiques des activités dédiées à la sauvegarde des préférences des utilisateurs.
— Il peut aussi arriver qu'on ait besoin d'écrire ou de lire des fichiers qui sont stockés sur le terminal ou sur un périphérique externe.

Ici, on ne parlera pas des bases de données. Mais bientôt, promis.

15.1. Préférences partagées

Utile voire indispensable pour un grand nombre d'applications, pouvoir enregistrer les paramètres des utilisateurs leur permettra de paramétrer de manière minutieuse vos applications de manière à ce qu'ils obtiennent le rendu qui convienne le mieux à leurs exigences.

15.1.1. Les données partagées

Le point de départ de la manipulation des préférences partagées est la classe `SharedPreferences` . Elle possède des méthodes permettant d'enregistrer et récupérer des paires de type identifiant-valeur pour les types de données primitifs, comme les entiers ou les chaînes de caractères. L'avantage réel étant bien sûr que ces données sont conservées même si l'application est arrêtée ou tuée. Ces préférences sont de plus accessibles depuis plusieurs composants au sein d'une même application.

Il existe trois façons d'avoir accès aux `SharedPreferences` :

— La plus simple est d'utiliser la méthode statique `SharedPreferences PreferenceManager.getDefaultSharedPreferences(Context context)`.
— Si vous désirez utiliser un fichier standard par activité, alors vous pourrez utiliser la méthode `SharedPreferences getPreferences(int mode)`.

— En revanche, si vous avez besoin de plusieurs fichiers que vous identifierez par leur nom, alors utilisez `SharedPreferences getSharedPreferences (String name, int mode)` où `name` sera le nom du fichier.

En ce qui concerne le second paramètre, `mode`, il peut prendre trois valeurs :

— `Context.MODE_PRIVATE`, pour que le fichier créé ne soit accessible que par l'application qui l'a créé.
— `Context.MODE_WORLD_READABLE`, pour que le fichier créé puisse être lu par n'importe quelle application.
— `Context.MODE_WORLD_WRITEABLE`, pour que le fichier créé puisse être lu *et* modifié par n'importe quelle application.

Petit détail, appeler `SharedPreferences PreferenceManager.getDefaultSharedPreferences(Context context)` revient à appeler `SharedPreferences getPreferences(MODE_PRIVATE)` et utiliser `SharedPreferences getPreferences(int mode)` revient à utiliser `SharedPreferences getSharedPreferences (NOM_PAR_DEFAUT, mode)` avec `NOM_PAR_DEFAUT` un nom généré en fonction du package de l'application.

Afin d'ajouter ou de modifier des couples dans un `SharedPreferences`, il faut utiliser un objet de type `SharedPreference.Editor` . Il est possible de récupérer cet objet en utilisant la méthode `SharedPreferences.Editor edit()` sur un `SharedPreferences`.

Si vous souhaitez ajouter des informations, utilisez une méthode du genre `SharedPreferences.Editor putX(String key, X value)` avec `X` le type de l'objet, `key` l'identifiant et `value` la valeur associée. Il vous faut ensuite impérativement valider vos changements avec la méthode `boolean commit()`.

Les préférences partagées ne fonctionnent qu'avec les objets de type `boolean`, `float`, `int`, `long` et `String`.

Par exemple, pour conserver la couleur préférée de l'utilisateur, il n'est pas possible d'utiliser la classe `Color` puisque seuls les types de base sont acceptés, alors on pourrait conserver la valeur de la couleur sous la forme d'une chaîne de caractères :

```
```

De manière naturelle, pour récupérer une valeur, on peut utiliser la méthode `X getX(String key, X defValue)` avec `X` le type de l'objet désiré, `key` l'identifiant de votre valeur et `defValue` une valeur que vous souhaitez voir retournée au cas où il n'y ait pas de valeur associée à `key` :

```
```

Si vous souhaitez supprimer une préférence, vous pouvez le faire avec `SharedPreferences.Editor removeString(String key)`, ou, pour radicalement supprimer toutes les préférences, il existe aussi `SharedPreferences.Editor clear()`.

Enfin, si vous voulez récupérer toutes les données contenues dans les préférences, vous pouvez utiliser la méthode `Map<String, ?> getAll()`.

15.1.2. Des préférences prêtes à l'emploi

Pour enregistrer vos préférences, vous pouvez très bien proposer une activité qui permet d'insérer différents paramètres (voir figure suivante). Si vous voulez développer vous-mêmes l'activité, grand bien vous fasse, ça fera des révisions, mais sachez qu'il existe un framework pour vous aider. Vous en avez sûrement déjà vus dans d'autres applications. C'est d'ailleurs un énorme avantage d'avoir toujours un écran similaire entre les applications pour la sélection des préférences.

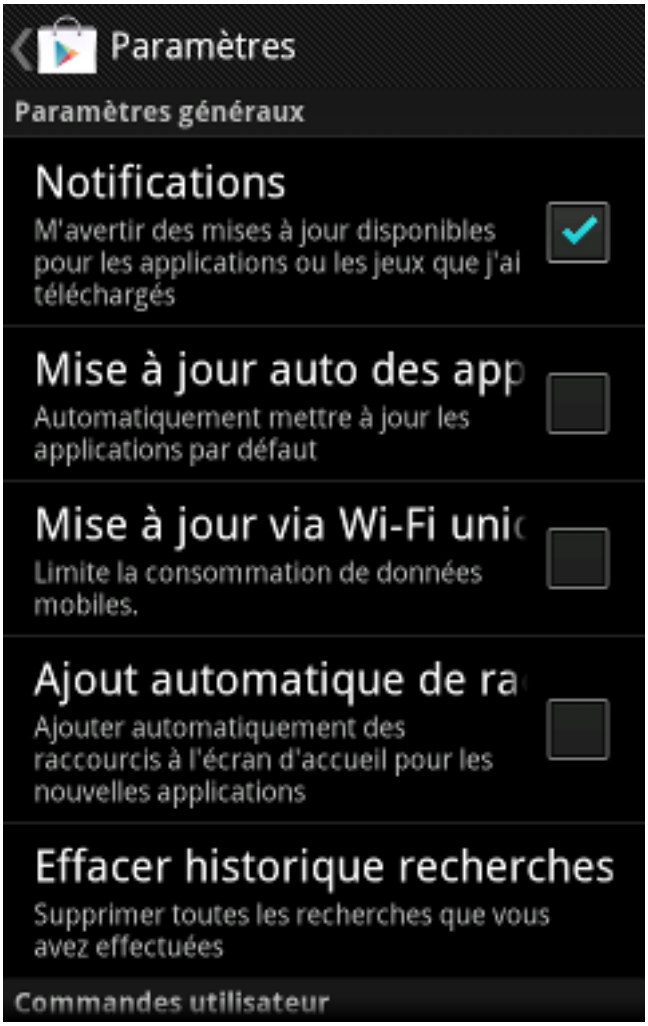

FIGURE 15.1. – L'activité permettant de choisir des paramètres pour le Play Store

Ce type d'activités s'appelle les « `PreferenceActivity` ». Un plus indéniable ici est que chaque couple identifiant/valeur est créé automatiquement et sera récupéré automatiquement, d'où un gain de temps énorme dans la programmation. La création se fait en plusieurs étapes, nous allons voir la première, qui consiste à établir une interface graphique en XML.

15.1.2.1. Étape 1 : en XML

La racine de ce fichier doit être un `PreferenceScreen`.

Comme ce n'est pas vraiment un layout, on le définit souvent dans `/xml/preference.xml`.

Tout d'abord, il est possible de désigner des catégories de préférences. Une pour les préférences destinées à internet par exemple, une autre pour les préférences esthétiques, etc. On peut ajouter des préférences avec le nœud `PreferenceCategory`. Ce nœud est un layout, il peut donc contenir d'autre vues. Il ne peut prendre qu'un seul attribut, `android:title`, pour préciser le texte qu'il affichera.

Ainsi le code suivant :

```
```

… donne le résultat visible à la figure suivante.

FIGURE 15.2. – Le code en image

Nous avons nos catégories, il nous faut maintenant insérer des préférences ! Ces trois vues ont cinq attributs en commun :

- `android:key` est l'identifiant de la préférence partagée. C'est un attribut *indispensable*, ne l'oubliez *jamais*.
- `android:title` est le titre principal de la préférence.
- `android:summary` est un texte secondaire qui peut être plus long et qui explique mieux ce que veut dire cette préférence.
- Utilisez `android:dependency`, si vous voulez lier votre préférence à une autre activité. Il faut y insérer l'identifiant `android:key` de la préférence dont on dépend.
- `android:defaultValue` est la valeur par défaut de cette préférence.

Il existe au moins trois types de préférences, la première étant une case à cocher avec `CheckBoxPreference`, avec `true` ou `false` comme valeur (soit la case est cochée, soit elle ne l'est pas).

À la place du résumé standard, vous pouvez déclarer un résumé qui ne s'affiche que quand la case est cochée, `android:summaryOn`, ou uniquement quand la case est décochée, `android:summaryOff`.

```
```

Ce qui donne la figure suivante.

FIGURE 15.3. – Regardez la première préférence, la case est cochée par défaut et c'est le résumé associé qui est affiché

Le deuxième type de préférences consiste à permettre à l'utilisateur d'insérer du texte avec `EditTextPreference`, qui ouvre une boîte de dialogue contenant un `EditText` permettant à l'utilisateur d'insérer du texte. On retrouve des attributs qui vous rappellerons fortement le chapitre sur les boîtes de dialogue. Par exemple, `android:dialogTitle` permet de définir le texte de la boîte de dialogue, alors que `android:negativeButtonText` et `android:positiveButtonText` permettent respectivement de définir le texte du bouton à droite et celui du bouton à gauche dans la boîte de dialogue.

Ce qui donne la figure suivante.

FIGURE 15.4. – Le code en image

De plus, comme vous avez pu le voir, ce paramètre est lié à la `CheckBoxPreference` précédente par l'attribut `android:dependency="checkBoxPref"`, ce qui fait qu'il ne sera accessible que si la case à cocher de `checkBoxPref` est activée, comme à la figure suivante.

FIGURE 15.5. – Le paramètre n'est accessible que si la case est cochée

De plus, comme nous l'avons fait avec les autres boîtes de dialogue, il est possible d'imposer un layout à l'aide de l'attribut `android:dialogLayout`.

Le troisième type de préférences est un choix dans une liste d'options avec `ListPreference`. Dans cette préférence, on différencie ce qui est affiché de ce qui est réel. Pratique pour traduire son application en plusieurs langues ! Encore une fois, il est possible d'utiliser les attributs `android:dialogTitle`, `android:negativeButtonText` et `android:positiveButtonText`. Les données de la liste que lira l'utilisateur sont à présenter dans l'attribut `android:entries`, alors que les données qui seront enregistrées sont à indiquer dans l'attribut `android:entryValues`. La manière la plus facile de remplir ces attributs se fait à l'aide d'une ressource de type `array`, par exemple pour la liste des couleurs :

```
```

Qu'on peut ensuite fournir aux attributs susnommés :

```
```

Ce qui donne la figure suivante.

FIGURE 15.6. – Le code en image

On a sélectionné « Vert », ce qui signifie que la valeur enregistrée sera `green`.

15.1.2.2. Étape 2 : dans le Manifest

Pour recevoir cette nouvelle interface graphique, nous avons besoin d'une activité. Il nous faut donc la déclarer dans le Manifest si on veut pouvoir y accéder avec les intents. Cette activité sera déclarée comme n'importe quelle activité :

```
```

15.1.2.3. Étape 3 : en Java

Notre activité sera en fait de type `PreferenceActivity`. On peut la traiter comme une activité classique, sauf qu'au lieu de lui assigner une interface graphique avec `setContentView`, on utilise `void addPreferencesFromResource(int preferencesResId)` en lui assignant notre layout :

15.2. Manipulation des fichiers

On a déjà vu comment manipuler certains fichiers précis à l'aide des ressources, mais il existe aussi des cas de figure où il faudra prendre en compte d'autres fichiers, par exemple dans le cas d'un téléchargement ou de l'exploration de fichiers sur la carte SD d'un téléphone. En théorie, vous ne serez pas très dépaysés ici puisqu'on manipule en majorité les mêmes méthodes qu'en Java. Il existe bien entendu quand même des différences.

Il y a deux manières d'utiliser les fichiers : soit sur la mémoire interne du périphérique à un endroit bien spécifique, soit sur une mémoire externe (par exemple une carte SD). Dans tous les cas, on part toujours du `Context` pour manipuler des fichiers.

15.2.1. Rappels sur l'écriture et la lecture de fichiers

Ce n'est pas un sujet forcément évident en Java puisqu'il existe beaucoup de méthodes qui permettent d'écrire et de lire des fichiers en fonction de la situation.

Le cas le plus simple est de manipuler des flux d'octets, ce qui nécessite des objets de type `FileInputStream` pour lire un fichier et `FileOutputStream` pour écrire dans un fichier. La lecture s'effectue avec la méthode `int read()` et on écrit dans un fichier avec `void write(byte[] b)`. Voici un programme très simple qui lit dans un fichier puis écrit dans un autre fichier :

15.2.2. En interne

L'avantage ici est que la présence des fichiers dépend de la présence de l'application. Par conséquent, les fichiers seront supprimés si l'utilisateur désinstalle l'activité. En revanche, comme la mémoire interne du téléphone risque d'être limitée, on évite en général de placer les plus gros fichiers de cette manière.

Afin de récupérer un `FileOutputStream` qui pointera vers le bon répertoire, il suffit d'utiliser la méthode `FileOutputStream openFileOutput (String name, int mode)` avec `name` le nom du fichier et `mode` le mode dans lequel ouvrir le fichier. Eh oui, encore une fois, il existe plusieurs méthodes pour ouvrir un fichier :

- `MODE_PRIVATE` permet de créer (ou de remplacer, d'ailleurs) un fichier qui sera utilisé uniquement par l'application.
- `MODE_WORLD_READABLE` pour créer un fichier que même d'autres applications pourront lire.
- `MODE_WORLD_WRITABLE` pour créer un fichier où même d'autres applications pourront lire et écrire.

— `MODE_APPEND` pour écrire à la fin d'un fichier préexistant, au lieu de créer un fichier.

Par exemple, pour écrire mon pseudo dans un fichier, je ferai :

```
```

De manière analogue, on peut retrouver un fichier dans lequel lire à l'aide de la méthode `FileInputStream openFileInput (String name)`.

N'essayez pas d'insérer des « / » ou des « \ » pour mettre vos fichiers dans un autre répertoire, sinon les méthodes renverront une exception.

Ensuite, il existe quelques méthodes qui permettent de manipuler les fichiers au sein de cet emplacement interne, afin d'avoir un peu plus de contrôle. Déjà, pour retrouver cet emplacement, il suffit d'utiliser la méthode `File getFilesDir()`. Pour supprimer un fichier, on peut faire appel à `boolean deleteFile(String name)` et pour récupérer une liste des fichiers créés par l'application, on emploie `String[] fileList()`.

15.2.2.1. Travailler avec le cache

Les fichiers normaux ne sont supprimés que si quelqu'un le fait, que ce soit vous ou l'utilisateur. *A contrario*, les fichiers sauvegardés avec le cache peuvent aussi être supprimés par le système d'exploitation afin de libérer de l'espace. C'est un avantage, pour les fichiers qu'on ne veut garder que temporairement.

Pour écrire dans le cache, il suffit d'utiliser la méthode `File getCacheDir()` pour récupérer le répertoire à manipuler. De manière générale, on évite d'utiliser trop d'espace dans le cache, il s'agit vraiment d'un espace temporaire de stockage pour petits fichiers.

Ne vous attendez pas forcément à ce qu'Android supprime les fichiers, il ne le fera que quand il en aura besoin, il n'y a pas de manière de prédire ce comportement.

15.2.3. En externe

Le problème avec le stockage externe, c'est qu'il n'existe aucune garantie que vos fichiers soient présents. L'utilisateur pourra les avoir supprimés ou avoir enlevé le périphérique de son emplacement. Cependant, cette fois la taille disponible de stockage est au rendez-vous ! Enfin, quand nous parlons de périphérique externe, nous parlons principalement d'une carte SD, d'une clé USB… ou encore d'un ordinateur !

Pour écrire sur un périphérique externe, il vous faudra ajouter la permission `WRITE_EXTERNAL_STORAGE`. Pour ce faire, il faut rajouter la ligne suivante à votre Manifest : `<uses-per`

`mission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />`.

Tout d'abord, pour vérifier que vous avez bien accès à la mémoire externe, vous pouvez utiliser la méthode statique `String Environment.getExternalStorageState()`. La chaîne de caractères retournée peut correspondre à plusieurs constantes, dont la plus importante reste `Environment.MEDIA_MOUNTED` pour savoir si le périphérique est bien monté et peut être lu (pour un périphérique bien monté mais qui ne peut pas être lu, on utilisera `Environment.MEDIA_MOUNTED_READ_ONLY`) :

```
```

Vous trouverez d'autres statuts à utiliser dans la documentation .

Afin d'obtenir la racine des fichiers du périphérique externe, vous pouvez utiliser la méthode statique `File Environment.getExternalStorageDirectory()`. Cependant, il est conseillé d'écrire des fichiers uniquement à un emplacement précis : `/Android/data/<votre_package>/files/`. En effet, les fichiers qui se trouvent à cet emplacement seront automatiquement supprimés dès que l'utilisateur effacera votre application.

15.2.3.1. Partager des fichiers

Il arrive aussi que votre utilisateur veuille partager la musique qu'il vient de concevoir avec d'autres applications du téléphone, pour la mettre en sonnerie par exemple. Ce sont des fichiers qui ne sont pas spécifiques à votre application ou que l'utilisateur ne souhaitera pas supprimer à la désinstallation de l'application. On va donc faire en sorte de sauvegarder ces fichiers à des endroits spécifiques. Une petite sélection de répertoires : pour la musique on mettra les fichiers dans `/Music/`, pour les téléchargements divers on utilisera `/Download/` et pour les sonneries de téléphone on utilisera `/Ringtones/`.

15.2.4. Application

15.2.4.1. Énoncé

Très simple, on va faire en sorte d'écrire votre pseudo dans deux fichiers : un en stockage interne, l'autre en stockage externe. N'oubliez pas de vérifier qu'il est possible d'écrire sur le support externe !

15.2.4.2. Détails techniques

Il existe une constante qui indique que le périphérique est en lecture seule (et que par conséquent il est impossible d'écrire dessus), c'est la constante `Environment.MEDIA_MOUNTED_READ_ONLY`.

Si un fichier n'existe pas, vous pouvez le créer avec `boolean createNewFile()`.

15.2.4.3. Ma solution

```
```

— Il est possible d'enregistrer les préférences de l'utilisateur avec la classe `SharedPreferences`.
— Pour permettre à l'utilisateur de sélectionner ses préférences, on peut définir une `PreferenceActivity` qui facilite le processus. On peut ainsi insérer des `CheckBox`, des `EditText`, etc.
— Il est possible d'enregistrer des données dans des fichiers facilement, comme en Java. Cependant, on trouve deux endroits accessibles : en interne (sur un emplacement mémoire réservé à l'application sur le téléphone) ou en externe (sur un emplacement mémoire amovible, comme par exemple une carte SD).
— Enregistrer des données sur le cache permet de sauvegarder des fichiers temporairement.

16. TP : un explorateur de fichiers

Petit à petit, on se rapproche d'un contenu qui pourrait s'apparenter à celui des applications professionnelles. Bien entendu, il nous reste du chemin à parcourir, mais on commence à vraiment voir comment fonctionne Android !

Afin de symboliser notre entrée dans les entrailles du système, on va s'affairer ici à déambuler dans ses méandres. Notre objectif : créer un petit explorateur qui permettra de naviguer entre les fichiers contenus dans le terminal et faire en sorte de pouvoir exécuter certains de ces fichiers.

16.1. Objectifs

16.1.1. Contenu d'un répertoire

L'activité principale affiche le contenu du répertoire dans lequel on se situe. Afin de différencier rapidement les fichiers des répertoires, ces derniers seront représentés avec une couleur différente. La figure suivante vous donne un avant-goût de ce que l'on obtiendra.

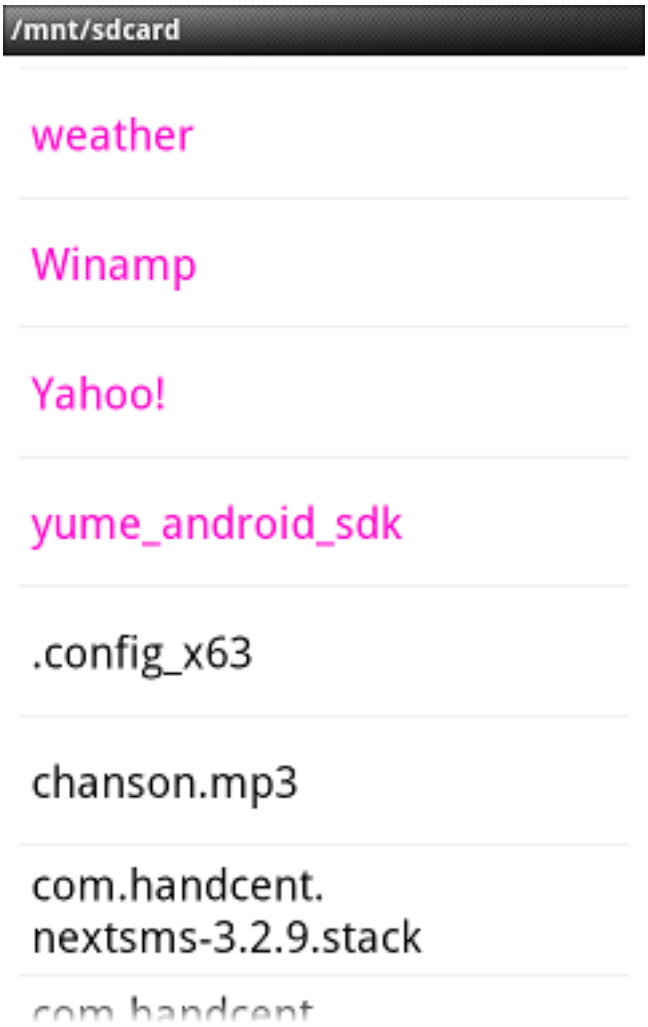

FIGURE 16.1. – Le dernier répertoire que contient le répertoire courant est « yume_android_sdk »

Notez aussi que le titre de l'activité change en fonction du répertoire dans lequel on se trouve. On voit sur la figure précédente que je me trouve dans le répertoire `sdcard`, lui-même situé dans `mnt`.

16.1.2. Navigation entre les répertoires

Si on clique sur un répertoire dans la liste, alors notre explorateur va entrer dedans et afficher la nouvelle liste des fichiers et répertoires. De plus, si l'utilisateur utilise le bouton `Retour Arrière`, alors il reviendra au répertoire parent du répertoire actuel. En revanche, si on se trouve à la racine de tous les répertoires, alors appuyer deux fois sur `Retour Arrière` fait sortir de l'application.

16.1.3. Préférences

Il faudra un menu qui permet d'ouvrir les préférences et où il sera possible de changer la couleur d'affichage des répertoires, comme à la figure suivante.

FIGURE 16.2. – L'application contiendra un menu de préférences

Cliquer sur cette option ouvre une boîte de dialogue qui permet de sélectionner la couleur voulue, comme le montre la figure suivante.

FIGURE 16.3. – Il sera possible de modifier la couleur d'affichage des répertoires

16.1.4. Action sur les fichiers

Cliquer sur un fichier fait en sorte de rechercher une application qui pourra le lire. Faire un clic long ouvre un menu contextuel qui permet soit de lancer le fichier comme avec un clic normal, soit de supprimer le fichier, ainsi que le montre la figure suivante.

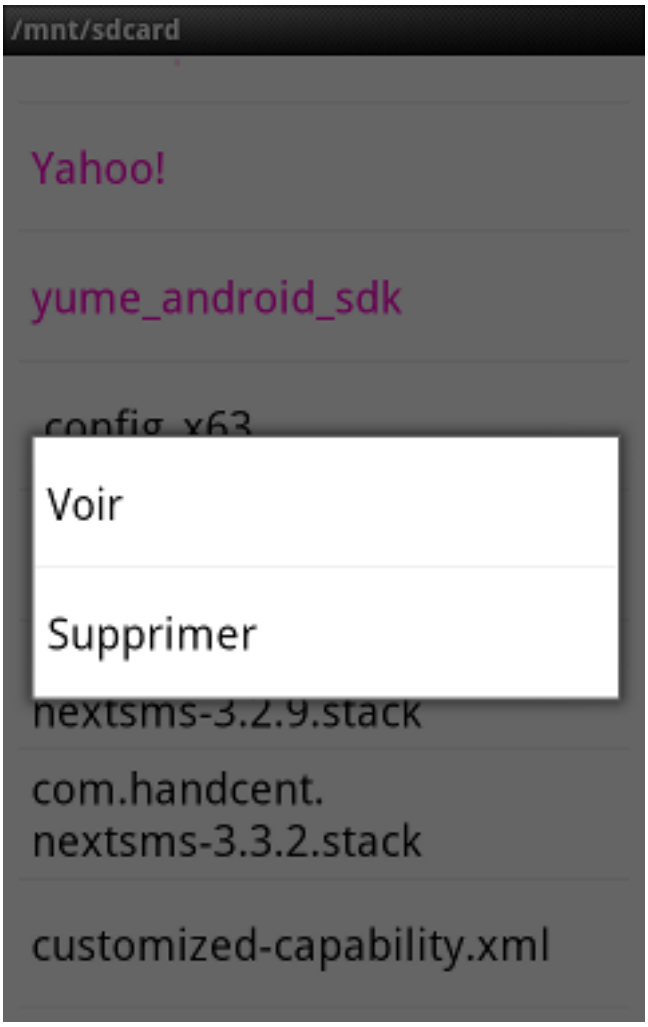

FIGURE 16.4. – Il est possible d'ouvrir ou de supprimer un fichier

Bien sûr, faire un clic long sur un répertoire ne propose pas d'exécuter ce dernier (on pourrait envisager de proposer de l'ouvrir, j'ai opté pour supprimer directement l'option).

16.2. Spécifications techniques

16.2.1. Activité principale

16.2.1.1. Un nouveau genre d'activité

La première chose à faire est de vérifier qu'il est possible de lire la carte SD avec les méthodes vues aux chapitres précédents. S'il est bien possible de lire la carte, alors on affiche la liste des fichiers du répertoire, ce qui se fera dans une `ListView`. Cependant, comme notre mise en page sera uniquement constituée d'une liste, nous allons procéder différemment par rapport à d'habitude. Au lieu d'avoir une activité qui affiche un layout qui contient une `ListView`, on va remplacer notre `Activity` par une `ListActivity`. Comme l'indique le nom, une `ListActivity` est une activité qui est principalement utilisée pour afficher une `ListView`. Comme il s'agit d'une classe qui dérive de `Activity`, il faut la traiter comme une activité normale, si ce n'est que vous

n'avez pas besoin de préciser un layout avec `void setContentView (View view)`, puisqu'on sait qu'il n'y a qu'une liste dans la mise en page. Elle sera alors ajoutée automatiquement.

Il est possible de récupérer la `ListView` qu'affiche la `ListActivity` à l'aide de la méthode `ListView getListView ()`. Cette `ListView` est une `ListView` tout à fait banale que vous pouvez traiter comme celles vues dans le cours.

16.2.1.2. Adaptateur personnalisé

On associera les items à la liste à l'aide d'un adaptateur personnalisé. En effet, c'est la seule solution pour avoir deux couleurs dans les éléments de la liste. On n'oubliera pas d'optimiser cet adaptateur afin d'avoir une liste fluide. Ensuite, on voudra que les éléments soient triés de la manière suivante :

— Les répertoires en premier, les fichiers en second.
— Dans chacune de ces catégories, les éléments sont triés dans l'ordre alphabétique sans tenir compte de la casse.

Pour cela, on pourra utiliser la méthode `void sort (Comparator<? super T> comparator)` qui permet de trier des éléments en fonction de règles qu'on lui passe en paramètres. Ces règles implémentent l'interface `Comparator` de manière à pouvoir définir *comment* seront triés les objets. Votre implémentation de cette interface devra redéfinir la méthode `int compare(T lhs, T rhs)` dont l'objectif est de dire qui est le plus grand entre `lhs` et `rsh`. Si `lhs` est plus grand que `rhs`, on renvoie un entier supérieur à 0, si `lhs` est plus petit que `rhs`, on renvoie un entier inférieur à 0, et s'ils sont égaux, on renvoie 0. Vous devrez vérifier que cette méthode respecte la logique suivante :

— `compare(a,a)` renvoie 0 pour tout `a` parce que `a==a`.
— `compare(a,b)` renvoie l'opposé de `compare(b,a)` pour toutes les paires (`a`,`b`) (par exemple, si `a > b`, alors `compare(a,b)` renvoie un entier supérieur à 0 et `compare(b,a)` un entier inférieur à 0).
— Si `compare(a,b) > 0` et `compare(b,c) > 0`, alors `compare(a,c) > 0` quelque que soit la combinaison (`a`, `b`, `c`).

Je comprends que ce soit un peu compliqué à comprendre, alors voici un exemple qui trie les entiers :

```
```

Ensuite, dans le code, on peut l'utiliser pour trier un tableau d'entiers :

```
```

16.2.2. Préférences

Nous n'avons qu'une préférence ici, qui chez moi a pour identifiant `repertoireColorPref` et qui contient la couleur dans laquelle nous souhaitons afficher les répertoires.

Comme il n'existe pas de vue qui permette de choisir une couleur, on va utiliser une vue développée par Google dans ses échantillons et qui n'est pas incluse dans le code d'Android. Tout ce qu'il faut faire, c'est créer un fichier Java qui s'appelle `ColorPickerView` et d'y insérer le code suivant :

```
```

Ce n'est pas grave si vous ne comprenez pas ce code compliqué, il permet juste d'afficher le joli rond de couleur et de sélectionner une couleur. En fait, la vue contient un listener qui s'appelle `OnColorChangedListener`. Ce listener se déclenche dès que l'utilisateur choisit une couleur. Afin de créer un objet de type `ColorPickerView`, on doit utiliser le constructeur `ColorPickerView(Context c, OnColorChangedListener listener, int color)` avec `listener` le listener qui sera déclenché dès qu'une couleur est choisie et `color` la couleur qui sera choisie par défaut au lancement de la vue.

Notre préférence, elle, sera une boîte de dialogue qui affichera ce `ColorPickerView`. Comme il s'agira d'une boîte de dialogue qui permettra de choisir une préférence, elle dérivera de `DialogPreference`.

Au moment de la construction de la boîte de dialogue, la méthode de *callback* `void onPrepareDialogBuilder(Builder builder)` est appelée, comme pour toutes les `AlertDialog`. On utilise `builder` pour construire la boîte, il est d'ailleurs facile d'y insérer une vue à l'aide de la méthode `AlertDialog.Builder setView(View view)`.

Notre préférence a un attribut de type `int` qui permet de retenir la couleur que choisit l'utilisateur. Elle peut avoir un attribut de type `OnColorChangedListener` ou implémenter elle-même `OnColorChangedListener`, dans tous les cas cette implémentation implique de redéfinir la fonction `void colorChanged(int color)` avec `color` la couleur qui a été choisie. Dès que l'utilisateur choisit une couleur, on change notre attribut pour désigner cette nouvelle couleur.

On n'enregistrera la bonne couleur qu'à la fermeture de la boîte de dialogue, celle-ci étant marquée par l'appel à la méthode `void onDialogClosed(boolean positiveResult)` avec `positiveResult` qui vaut `true` si l'utilisateur a cliqué sur `OK`.

16.2.2.1. Réagir au changement de préférence

Dès que l'utilisateur change de couleur, il faudrait que ce changement se répercute immédiatement sur l'affichage des répertoires. Il nous faut donc détecter les changements de configuration. Pour cela, on va utiliser l'interface `OnSharedPreferenceChangeListener`. Cette interface fait appel à la méthode de *callback* `void onSharedPreferenceChanged(SharedPreferences sharedPreferences, String key)` dès qu'un changement de préférence arrive, avec `sharedPreferences` l'ensemble des préférences et `key` la clé de la préférence qui vient d'être modifiée. On peut indiquer à `SharedPreferences` qu'on souhaite ajouter un listener à l'aide de la

méthode `void registerOnSharedPreferenceChangeListener (SharedPreferences.On SharedPreferenceChangeListener listener)`.

16.2.3. Options

Ouvrir le menu d'options ne permet d'accéder qu'à une option. Cliquer sur celle-ci enclenche un intent explicite qui ouvrira la `PreferenceActivity`.

16.2.4. Navigation

Il est recommandé de conserver un `File` qui représente le répertoire courant. On peut savoir si un fichier est un répertoire avec la méthode `boolean isDirectory()` et, s'il s'agit d'un répertoire, on peut voir la liste des fichiers qu'il contient avec `File[] listFiles()`.

Pour effectuer des retours en arrière, il faut détecter la pression du bouton adéquat. À chaque fois qu'on presse un bouton, la méthode de *callback* `boolean onKeyDown(int keyCode, KeyEvent event)` est lancée, avec `keyCode` un code qui représente le bouton pressé et `event` l'évènement qui s'est produit. Le code du bouton `Retour arrière` est `KeyEvent.KEYCODE_BACK`.

Il existe deux cas pour un retour en arrière :

— Soit on ne se trouve pas à la racine de la hiérarchie de fichier, auquel cas on peut revenir en arrière dans cette hiérarchie. Il faut passer au répertoire parent du répertoire actuel et ce répertoire peut se récupérer avec la méthode `File getParentFile()`.
— Soit on se trouve à la racine et il n'est pas possible de faire un retour en arrière. En ce cas, on propose à l'utilisateur de quitter l'application avec la méthode de `Context` que vous connaissez déjà, `void finish()`.

16.2.5. Visualiser un fichier

Nous allons bien entendu utiliser des intents implicites qui auront pour action `ACTION_VIEW`. Le problème est de savoir comment associer un type et une donnée à un intent, depuis un fichier. Pour la donnée, il existe une méthode statique de la classe `Uri` qui permet d'obtenir l'URI d'un fichier : `Uri.fromFile(File file)`. Pour le type, c'est plus délicat. Il faudra détecter l'extension du fichier pour associer un type qui corresponde. Par exemple, pour un fichier `.mp3`, on indiquera le type MIME `audio/mp3`. Enfin, si on veut moins s'embêter, on peut aussi passer le type MIME `audio/*` pour chaque fichier audio.

Pour rajouter une donnée *et* un type en même temps à un intent, on utilise la méthode `void setDataAndType(Uri data, String type)`, car, si on utilise la méthode `void setData(Uri)`, alors le champ `type` de l'intent est supprimé, et si on utilise `void setType(String)`, alors le champ `data` de l'intent est supprimé. Pour récupérer l'extension d'un fichier, il suffit de récupérer son nom avec `String getName()`, puis de récupérer une partie de ce nom : toute la partie qui se trouve après le point qui représente l'extension :

`int indexOf(String str)` va trouver l'endroit où se trouve la première instance de `str` dans la chaîne de caractères, alors que `String substring(int beginIndex)` va extraire la sous-chaîne de caractères qui se situe à partir de `beginIndex` jusqu'à la fin de cette chaîne. Donc, si le fichier s'appelle `chanson.mp3`, la position du point est 7 (puisqu'on commence à 0), on prend donc la sous-chaîne à partir du caractère 8 jusqu'à la fin, ce qui donne « `mp3` ». C'est la même chose que si on avait fait :

N'oubliez pas de gérer le cas où vous n'avez pas d'activité qui puisse intercepter votre intent.

16.3. Ma solution

16.3.1. Interface graphique

Facile, il n'y en a pas ! Comme notre activité est constituée uniquement d'une `ListView`, pas besoin de lui attribuer une interface graphique avec `setContentView`.

16.3.2. Choisir une couleur avec ColorPickerPreferenceDialog

Tout le raisonnement a déjà été expliqué dans les spécifications techniques :

Il faut ensuite ajouter cette boîte de dialogue dans le fichier XML des préférences :

Il suffit ensuite de déclarer l'activité dans le Manifest :

… puis de créer l'activité :

16.3.3. L'activité principale

16.3.3.1. Attributs

Voici les différents attributs que j'utilise :

```
```

Comme je fais implémenter `OnSharedPreferenceChangeListener` à mon activité, je dois redéfinir la méthode de *callback* :

```
```

16.3.3.2. L'adaptateur

J'utilise un `Adapter` que j'ai créé moi-même afin d'avoir des items de la liste de différentes couleurs :

```
```

16.3.3.3. Méthodes secondaires

Ensuite, j'ai une méthode qui permet de vider l'adaptateur :

```
```

J'ai aussi développé une méthode qui me permet de passer d'un répertoire à l'autre :

```
```

Cette méthode est d'ailleurs utilisée par la méthode de *callback* `onKeyDown` :

```
```

16.3.3.4. Gestion de l'intent pour visualiser un fichier

```
```

16.3.3.5. Les menus

Rien d'étonnant ici, normalement vous connaissez déjà tout. À noter que j'ai utilisé deux layouts pour le menu contextuel de manière à pouvoir le changer selon qu'il s'agit d'un répertoire ou d'un fichier :

16.3.3.6. onCreate

Voici la méthode principale où se situent toutes les initialisations :

Télécharger le projet

16.4. Améliorations envisageables

16.4.1. Quand la liste est vide ou le périphérique externe est indisponible

On se trouve en face d'un écran blanc pas très intéressant… Ce qui pourrait être plus excitant, c'est un message qui indique à l'utilisateur qu'il n'a pas accès à ce périphérique externe. On peut faire ça en indiquant un layout pour notre `ListActivity` ! Oui, je sais, je vous ai dit de ne pas le faire, parce que notre activité contient principalement une liste, mais là on pousse le concept encore plus loin. Le layout qu'on utilisera doit contenir au moins une `ListView` pour représenter celle de notre `ListActivity`, mais notre application sera bien incapable de la trouver si vous ne lui précisez pas où elle se trouve. Vous pouvez le faire en mettant comme identifiant à la `ListView` `android:id="@android:id/list"`. Si vous voulez qu'un widget ou un layout s'affiche quand la liste est vide, vous devez lui attribuer l'identifiant `android:id="@android:id/empty"`. Pour ma correction, j'ai le XML suivant :

16.4.2. Détection automatique du type MIME

Parce que faire une longue liste de « Si on a cette extension pour ce fichier, alors le type MIME, c'est celui-là » est quand même long et contraignant, je vous propose de détecter automatiquement le type MIME d'un objet. Pour cela, on utilisera un objet de type `MimeTypeMap`. Afin de récupérer cet objet, on passe par la méthode statique `MimeTypeMap` `MimeTypeMap.getSingleton()`.

Petite digression pour vous dire que le *design pattern* singleton a pour objectif de faire en sorte que vous ne puissiez avoir qu'une seule instance d'un objet. C'est pourquoi on utilise la méthode `getSingleton()` qui renvoie toujours le même objet. Il est impossible de construire autrement un objet de type `MimeTypeMap`.

Ensuite c'est simple, il suffit de donner à la méthode String `getMimeTypeFromExtension(String extension)` l'extension de notre fichier. On obtient ainsi :

16.4.3. Détecter les changements d'état du périphérique externe

C'est bien beau tout ça, mais si l'utilisateur se décide tout à coup à changer la carte SD en pleine utilisation, nous ferons face à un gros plantage ! Alors comment contrer ce souci ? C'est simple. Dès que l'état du périphérique externe change, un broadcast intent est transmis pour le signaler à tout le système. Il existe tout un tas d'actions différentes associées à un changement d'état, je vous propose de ne gérer que le cas où le périphérique externe est enlevé, auquel cas l'action est `ACTION_MEDIA_REMOVED`. Notez au passage que l'action pour dire que la carte fonctionne à nouveau est `ACTION_MEDIA_MOUNTED`.

Comme nous l'avons vu dans le cours, il faudra déclarer notre `broadcast receiver` dans le Manifest :

Ensuite, dans le `receiver` en lui-même, on fait en sorte de viser la liste des éléments s'il y a un problème avec le périphérique externe, ou au contraire de la repeupler dès que le périphérique fonctionne correctement à nouveau. À noter que dans le cas d'un `broadcast Intent` avec l'action `ACTION_MEDIA_MOUNTED`, l'intent aura dans son champ `data` l'emplacement de la racine du périphérique externe :

17. Les bases de données

Ce que nous avons vu précédemment est certes utile, mais ne répondra pas à tous nos besoins. Nous avons besoin d'un moyen efficace de stocker des données complexes et d'y accéder. Or, il nous faudrait des années pour concevoir un système de ce style. Imaginez le travail s'il vous fallait développer de A à Z une bibliothèque multimédia qui puisse chercher en moins d'une seconde parmi plus de 100 000 titres une chanson bien précise ! C'est pour cela que nous avons besoin des bases de données, qui sont optimisées pour ce type de traitements.

Les bases de données pour Android sont fournies à l'aide de SQLite . L'avantage de SQLite est qu'il s'agit d'un SGBD très compact et par conséquent très efficace pour les applications embarquées, mais pas uniquement puisqu'on le trouve dans Skype, Adobe Reader, Firefox, etc.

17.1. Généralités

Vous comprendrez peut-être ce chapitre même si vous n'avez jamais manipulé de bases de données auparavant. Tant mieux, mais cela ne signifie pas que vous serez capables de manipuler correctement les bases de données pour autant. C'est une vraie science que d'agencer les bases de données et il faut beaucoup plus de théorie que nous n'en verrons ici pour modéliser puis réaliser une base de données cohérente et efficace.

Il vous est possible d'apprendre à utiliser les bases de données et surtout MySQL grâce au cours « Administrez vos bases de données avec MySQL » rédigé par Taguan .

17.1.1. Sur les bases de données

Une base de données est un dispositif permettant de stocker un ensemble d'informations de manière structurée. L'agencement adopté pour organiser les informations s'appelle le **schéma**.

L'unité de base de cette structure s'appelle la **table**. Une table regroupe des ensembles d'informations qui sont composés de manière similaire. Une entrée dans une table s'appelle un **enregistrement,** ou un **tuple**. Chaque entrée est caractérisée par plusieurs renseignements distincts, appelés des **champs** ou **attributs**.

Par exemple, une table peut contenir le prénom, le nom et le métier de plusieurs utilisateurs, on aura donc pour chaque utilisateur les mêmes informations. Il est possible de représenter une table par un tableau, où les champs seront symbolisés par les colonnes du tableau et pour lequel chaque ligne représentera une entrée différente. Regardez la figure suivante, cela devrait être plus clair.

Nom	Prénom	Métier
Daku	Tenshi	Auteur
John	John	Éditeur
Andro	Wiiid	Validateur
Shakespeare	William	Auteur
...	...	...

FIGURE 17.1. – Cette table contient quatre tuples qui renseignent toutes des informations du même gabarit pour chaque attribut

Une manière simple d'identifier les éléments dans une table est de leur attribuer une **clé**. C'est-à-dire qu'on va choisir une combinaison de champs qui permettront de récupérer de manière unique un enregistrement. Dans la table présentée à la figure suivante, l'attribut `Nom` peut être une clé puisque toutes les entrées ont un `Nom` différent. Le problème est qu'il peut arriver que deux utilisateurs aient le même nom, c'est pourquoi on peut aussi envisager la combinaison `Nom` et `Prénom` comme clé.

Nom	**Prénom**	Métier
Daku	Tenshi	Auteur
John	John	Éditeur
Andro	Wiiid	Validateur
Shakespeare	William	Auteur
...	...	...

FIGURE 17.2. – On choisit comme clé la combinaison Nom-Prénom

Il n'est pas rare qu'une base de données ait plusieurs tables. Afin de lier des tables, il est possible d'insérer dans une table une clé qui appartient à une autre table, auquel cas on parle de **clé étrangère** pour la table qui accueille la clé, comme à la figure suivante.

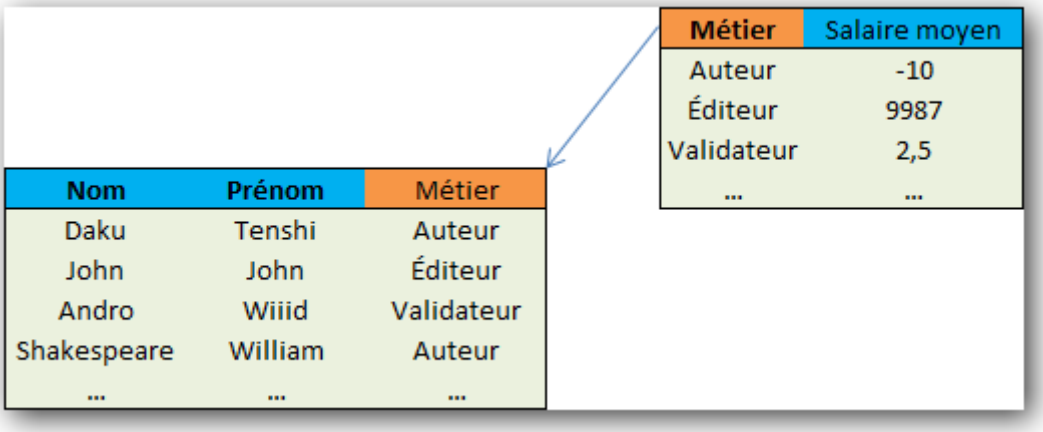

FIGURE 17.3. – Dans notre première table, Métier est une clé étrangère, car elle est clé primaire de la seconde table

Il est possible d'effectuer des opérations sur une base de données, comme créer des tables, supprimer des entrées, etc. L'opération qui consiste à lire des informations qui se trouvent dans une base de données s'appelle la **sélection**.

Pour effectuer des opérations sur plusieurs tables, on passe par une **jointure**, c'est-à-dire qu'on combine des attributs qui appartiennent à plusieurs tables pour les présenter conjointement.

Afin d'effectuer toutes ces opérations, on passe par un **langage de requête**. Celui dont nous avons besoin s'appelle **SQL**. Nous verrons un rappel des opérations principales dans ce chapitre.

Enfin, une base de données est destinée à recueillir des informations simples, c'est pourquoi on évite d'y insérer des données lourdes comme des fichiers ou des données brutes. Au lieu de mettre directement des images ou des vidéos, on préfèrera insérer un URI qui dirige vers ces fichiers.

17.1.2. Sur SQLite

Contrairement à MySQL par exemple, SQLite ne nécessite pas de serveur pour fonctionner, ce qui signifie que son exécution se fait dans le même processus que celui de l'application. Par conséquent, une opération massive lancée dans la base de données aura des conséquences visibles sur les performances de votre application. Ainsi, il vous faudra savoir maîtriser son implémentation afin de ne pas pénaliser le restant de votre exécution.

17.1.3. Sur SQLite pour Android

SQLite a été inclus dans le cœur même d'Android, c'est pourquoi chaque application peut avoir sa propre base. De manière générale, les bases de données sont stockées dans les répertoires de la forme `/data/data/<package>/databases`. Il est possible d'avoir plusieurs bases de données par application, cependant chaque fichier créé l'est selon le mode `MODE_PRIVATE`, par conséquent les bases ne sont accessibles qu'au sein de l'application elle-même. Notez que ce n'est pas pour autant qu'une base de données ne peut pas être partagée avec d'autres applications.

Enfin, pour des raisons qui seront expliquées dans un chapitre ultérieur, il est préférable de faire en sorte que la clé de chaque table soit un identifiant qui s'incrémente automatiquement. Notre schéma devient donc la figure suivante.

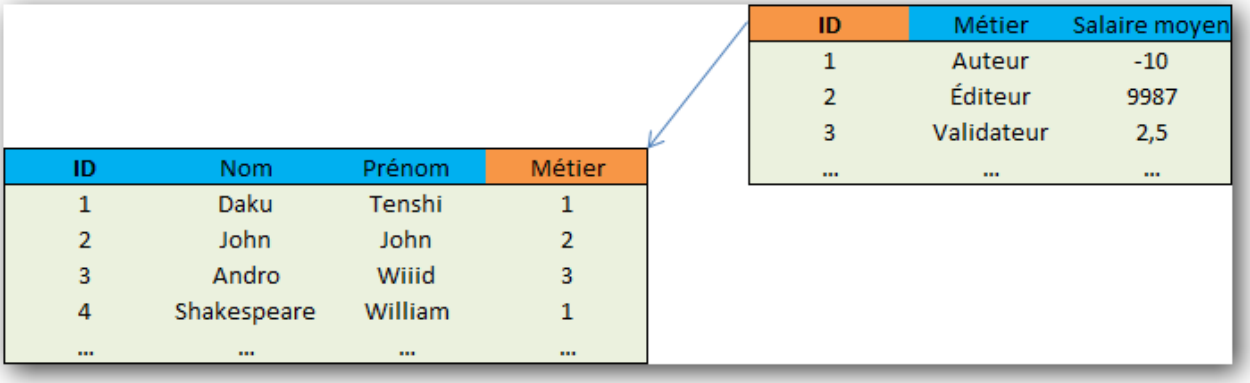

FIGURE 17.4. – Un identifiant a été ajouté

17.2. Création et mise à jour

La solution la plus évidente est d'utiliser une classe qui nous aidera à maîtriser toutes les relations avec la base de données. Cette classe dérivera de `SQLiteOpenHelper`. Au moment de la création de la base de données, la méthode de *callback* `void onCreate(SQLiteDatabase db)` est automatiquement appelée, avec le paramètre `db` qui représentera la base. C'est dans cette méthode que vous devrez lancer les instructions pour créer les différentes tables et éventuellement les remplir avec des données initiales.

Pour créer une table, il vous faudra réfléchir à son nom et à ses attributs. Chaque attribut sera défini à l'aide d'un type de données. Ainsi, dans la table `Metier` de notre exemple, nous avons trois attributs :

- `ID`, qui est un entier auto-incrémental pour représenter la clé ;
- `Métier`, qui est une chaîne de caractères ;
- `Salaire`, qui est un nombre réel.

Pour SQLite, c'est simple, il n'existe que cinq types de données :

- `NULL` pour les données `NULL`.
- `INTEGER` pour les entiers (sans virgule).
- `REAL` pour les nombres réels (avec virgule).
- `TEXT` pour les chaînes de caractères.
- `BLOB` pour les données brutes, par exemple si vous voulez mettre une image dans votre base de données (ce que vous ne ferez jamais, n'est-ce pas ?).

La création de table se fait avec une syntaxe très naturelle :

```
```

Pour chaque attribut, on doit déclarer au moins deux informations :

- Son nom, afin de pouvoir l'identifier ;
- Son type de donnée.

Mais il est aussi possible de déclarer des contraintes pour chaque attribut à l'emplacement de `{contraintes}`. On trouve comme contraintes :

- `PRIMARY KEY` pour désigner la clé primaire sur un attribut ;
- `NOT NULL` pour indiquer que cet attribut ne peut valoir `NULL` ;
- `CHECK` afin de vérifier que la valeur de cet attribut est cohérente ;
- `DEFAULT` sert à préciser une valeur par défaut.

Ce qui peut donner par exemple :

```
```

Il existe deux types de requêtes SQL. Celles qui appellent une réponse, comme la sélection, et celles qui n'appellent pas de réponse. Afin d'exécuter une requête SQL pour laquelle on ne souhaite pas de réponse ou on ignore la réponse, il suffit d'utiliser la méthode `void execSQL(String sql)`. De manière générale, on utilisera `execSQL(String)` dès qu'il ne s'agira pas de faire un `SELECT`, `UPDATE`, `INSERT` ou `DELETE`. Par exemple, pour notre table `Metier` :

```
```

Comme vous l'aurez remarqué, une pratique courante avec la manipulation des bases de données est d'enregistrer les attributs, tables et requêtes dans des constantes de façon à les retrouver et les modifier plus facilement. Tous ces attributs sont `public` puisqu'il est possible qu'on manipule la base en dehors de cette classe.

Si vous installez la base de données sur un périphérique externe, il vous faudra demander la permission `WRITE_EXTERNAL_STORAGE`, sinon votre base de données sera en lecture seule. Vous pouvez savoir si une base de données est en lecture seule avec la méthode `boolean isReadOnly()`.

Le problème du code précédent, c'est qu'il ne fonctionnera pas, et ce pour une raison très simple : il faut aussi implémenter la méthode `void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)` qui est déclenchée à chaque fois que l'utilisateur met à jour son application. `oldVersion` est le numéro de l'ancienne version de la base de données que l'application utilisait, alors que `newVersion` est le numéro de la nouvelle version. En fait, Android rajoute automatiquement dans la base une table qui contient la dernière valeur connue de la base. À chaque lancement, Android vérifiera la dernière version de la base par rapport à la version actuelle dans le code. Si le numéro de la version actuelle est supérieur à celui de la dernière version, alors cette méthode est lancée.

En général, le contenu de cette méthode est assez constant puisqu'on se contente de supprimer les tables déjà existantes pour les reconstruire suivant le nouveau schéma :

```
```

17.3. Opérations usuelles

17.3.1. Récupérer la base

Si vous voulez accéder à la base de données n'importe où dans votre code, il vous suffit de construire une instance de votre `SQLiteOpenHelper` avec le constructeur `SQLiteOpenHelper(Context context, String name, SQLiteDatabase.CursorFactory factory, int version)`, où `name` est le nom de la base, `factory` est un paramètre qu'on va oublier pour l'instant — qui accepte très bien les `null` — et `version` la version voulue de la base de données.

On utilise `SQLiteDatabase getWritableDatabase()` pour récupérer ou créer une base sur laquelle vous voulez lire et/ou écrire. La dernière méthode qui est appelée avant de fournir la base à `getWritableDatabase()` est la méthode de *callback* `void onOpen(SQLiteDatabase db)`, c'est donc l'endroit où vous devriez effectuer des opérations si vous le souhaitez.

Cependant, le système fera appel à une autre méthode avant d'appeler `onOpen(SQLiteDatabase)`. Cette méthode dépend de l'état de la base et de la version qui a été fournie à la création du `SQLiteOpenHelper` :

— S'il s'agit de la première fois que vous appelez la base, alors ce sera la méthode `onCreate(SQLiteDatabase)` qui sera appelée.
— Si la version fournie est plus récente que la dernière version fournie, alors on fait appel à `onUpgrade(SQLiteDatabase, int, int)`.
— En revanche, si la version fournie est plus ancienne, on considère qu'on effectue un retour en arrière et c'est `onDowngrade(SQLiteDatabase, int, int)` qui sera appelée.

L'objet `SQLiteDatabase` fourni est un objet en cache, constant. Si des opérations se déroulent sur la base après que vous avez récupéré cet objet, vous ne les verrez pas sur l'objet. Il faudra en recréer un pour avoir le nouvel état de la base.

Vous pouvez aussi utiliser la méthode `SQLiteDatabase getReadableDatabase()` pour récupérer la base, la différence étant que la base sera en lecture seule, mais uniquement s'il y a un problème qui l'empêche de s'ouvrir normalement. Avec `getWritableDatabase()`, si la base ne peut pas être ouverte en écriture, une exception de type `SQLiteException` sera lancée. Donc, si vous souhaitez ne faire que lire dans la base, utilisez en priorité `getReadableDatabase()`.

Ces deux méthodes peuvent prendre du temps à s'exécuter.

Enfin, il faut fermer une base comme on ferme un flux avec la méthode `void close()`.

Récupérer un `SQLiteDatabase` avec l'une des deux méthodes précédentes équivaut à faire un `close()` sur l'instance précédente de `SQLiteDatabase`.

17.3.2. Réfléchir, puis agir

Comme je l'ai déjà dit, chacun fait ce qu'il veut dès qu'il doit manipuler une base de données, ce qui fait qu'on se retrouve parfois avec du code incompréhensible ou difficile à mettre à jour. Une manière efficace de gérer l'interfaçage avec une base de données est de passer par un DAO, un objet qui incarne l'accès aux données de la base.

En fait, cette organisation implique d'utiliser deux classes :

— Une classe (dans mon cas `Metier`) qui représente les informations et qui peut contenir un enregistrement d'une table. Par exemple, on aura une classe `Metier` pour symboliser les différentes professions qui peuvent peupler cette table.
— Une classe `contrôleur`, le DAO pour ainsi dire, qui effectuera les opérations sur la base.

17.3.2.1. La classe Metier

Très simple, il suffit d'avoir un attribut pour chaque attribut de la table et d'ajouter des méthodes pour y accéder et les modifier :

17.3.2.2. La classe DAO

On doit y inclure au moins les méthodes CRUD, autrement dit les méthodes qui permettent l'ajout d'entrées dans la base, la récupération d'entrées, la mise à jour d'enregistrements ou encore la suppression de tuples. Bien entendu, ces méthodes sont à adapter en fonction du contexte et du métier. De plus, on rajoute les constantes globales déclarées précédemment dans la base :

Il ne s'agit bien entendu que d'un exemple, dans la pratique on essaie de s'adapter au contexte quand même, là je n'ai mis que des méthodes génériques.

Comme ces opérations se déroulent sur la base, nous avons besoin d'un accès à la base. Pour cela, et comme j'ai plusieurs tables dans mon schéma, j'ai décidé d'implémenter toutes les méthodes qui permettent de récupérer ou de fermer la base dans une classe abstraite :

Ainsi, pour pouvoir utiliser ces méthodes, la définition de ma classe `MetierDAO` devient :

17.3.3. Ajouter

Pour ajouter une entrée dans la table, on utilise la syntaxe suivante :

La partie `(nom_de_la_colonne1, nom_de_la_colonne2, …)` permet d'associer une valeur à une colonne précise à l'aide de la partie `(valeur1, valeur2, …)`. Ainsi la colonne 1 aura la valeur 1 ; la colonne 2, la valeur 2 ; etc.

Pour certains SGBD, l'instruction suivante est aussi possible afin d'insérer une entrée vide dans la table.

```
```

Cependant, avec SQLite ce n'est pas possible, il faut préciser au moins une colonne, quitte à lui passer comme valeur `NULL`.

```
```

En Java, pour insérer une entrée, on utilisera la méthode `long insert(String table, String nullColumnHack, ContentValues values)`, qui renvoie le numéro de la ligne ajoutée où :

— `table` est l'identifiant de la table dans laquelle insérer l'entrée.
— `nullColumnHack` est le nom d'une colonne à utiliser au cas où vous souhaiteriez insérer une entrée vide. Prenez n'importe laquelle.
— `values` est un objet qui représente l'entrée à insérer.

Les `ContentValues` sont utilisés pour insérer des données dans la base. Ainsi, on peut dire qu'ils fonctionnent un peu comme les `Bundle` par exemple, puisqu'on peut y insérer des couples identifiant-valeur, qui représenteront les attributs des objets à insérer dans la base. L'identifiant du couple doit être une chaîne de caractères qui représente une des colonnes de la table visée. Ainsi, pour insérer le métier « Caricaturiste », il me suffit de faire :

```
```

Je n'ai pas besoin de préciser de valeur pour l'identifiant puisqu'il s'incrémente tout seul.

17.3.4. Supprimer

La méthode utilisée pour supprimer est quelque peu différente. Il s'agit de `int delete(String table, String whereClause, String[] whereArgs)`. L'entier renvoyé est le nombre de lignes supprimées. Dans cette méthode :

— `table` est l'identifiant de la table.
— `whereClause` correspond au `WHERE` en SQL. Par exemple, pour sélectionner la première valeur dans la table `Metier`, on mettra pour `whereClause` la chaîne « `id = 1` ». En pratique, on préférera utiliser la chaîne « `id = ?` » et je vais vous expliquer pourquoi tout de suite.
— `whereArgs` est un tableau des valeurs qui remplaceront les « ? » dans `whereClause`. Ainsi, si `whereClause` vaut « `LIKE ? AND salaire > ?` » et qu'on cherche les métiers qui ressemblent à « ingénieur avec un salaire supérieur à 1000 € », il suffit d'insérer dans `whereArgs` un `String[]` du genre `{"ingenieur", "1000"}`.

Ainsi dans notre exemple, pour supprimer une seule entrée, on fera :

17.3.5. Mise à jour

Rien de très surprenant ici, la syntaxe est très similaire à la précédente : `int update(String table, ContentValues values, String whereClause, String[] whereArgs)`. On ajoute juste le paramètre `values` pour représenter les changements à effectuer dans le ou les enregistrements cibles. Donc, si je veux mettre à jour le salaire d'un métier, il me suffit de mettre à jour l'objet associé et d'insérer la nouvelle valeur dans un `ContentValues` :

17.3.6. Sélection

Ici en revanche, la méthode est plus complexe et revêt trois formes différentes en fonction des paramètres qu'on veut lui passer. La première forme est celle-ci :

La deuxième forme s'utilise sans l'attribut `limit` et la troisième sans les attributs `limit` *et* `distinct`. Ces paramètres sont vraiment explicites puisqu'ils représentent à chaque fois des mots-clés du SQL ou des attributs que nous avons déjà rencontrés. Voici leur signification :

- `distinct`, si vous ne voulez pas de résultats en double.
- `table` est l'identifiant de la table.
- `columns` est utilisé pour préciser les colonnes à afficher.
- `selection` est l'équivalent du `whereClause` précédent.
- `selectionArgs` est l'équivalent du `whereArgs` précédent.
- `group by` permet de grouper les résultats.
- `having` est utile pour filtrer parmi les groupes.
- `order by` permet de trier les résultats. Mettre `ASC` pour trier dans l'ordre croissant et `DESC` pour l'ordre décroissant.
- `limit` pour fixer un nombre maximal de résultats voulus.

Pour être franc, utiliser ces méthodes m'agace un peu, c'est pourquoi je préfère utiliser `Cursor rawQuery(String sql, String[] selectionArgs)` où je peux écrire la requête que je veux dans `sql` et remplacer les « ? » dans `selectionArgs`. Ainsi, si je veux tous les métiers qui rapportent en moyenne plus de 1€, je ferai :

17.4. Les curseurs

17.4.1. Manipuler les curseurs

Les curseurs sont des objets qui contiennent les résultats d'une recherche dans une base de données. Ce sont en fait des objets qui fonctionnent comme les tableaux que nous avons vus précédemment, ils contiennent les colonnes et lignes qui ont été renvoyées par la requête.

Ainsi, pour la requête suivante sur notre table `Metier` :

```
```

… on obtient le résultat visible à la figure suivante, dans un curseur.

ID	Métier	Salaire moyen
1	Auteur	-10
2	Éditeur	9987
3	Validateur	2,5

FIGURE 17.5. – On a trois lignes et trois colonnes

17.4.1.1. Les lignes

Ainsi, pour parcourir les résultats d'une requête, il faut procéder ligner par ligne. Pour naviguer parmi les lignes, on peut utiliser les méthodes suivantes :

— `boolean moveToFirst()` pour aller à la première ligne.
— `boolean moveToLast()` pour aller à la dernière.
— `boolean moveToPosition(int position)` pour aller à la `position` voulue, sachant que vous pouvez savoir le nombre de lignes avec la méthode `int getCount()`.

Cependant, il y a mieux. En fait, un `Cursor` est capable de retenir la position du dernier élément que l'utilisateur a consulté, il est donc possible de naviguer d'avant en arrière parmi les lignes grâce aux méthodes suivantes :

— `boolean moveToNext()` pour aller à la ligne suivante. Par défaut on commence à la ligne -1, donc, en utilisant un `moveToNext()` sur un tout nouveau `Cursor`, on passe à la première ligne. On aurait aussi pu accéder à la première ligne avec `moveToFirst()`, bien entendu.
— `boolean moveToPrevious()` pour aller à l'entrée précédente.

Vous remarquerez que toutes ces méthodes renvoient des booléens. Ces booléens valent `true` si l'opération s'est déroulée avec succès, sinon `false` (auquel cas la ligne demandée n'existe pas).

Pour récupérer la position actuelle, on utilise `int getPosition()`. Vous pouvez aussi savoir si vous êtes après la dernière ligne avec `boolean isAfterLast()`.

Par exemple, pour naviguer entre toutes les lignes d'un curseur, on fait :

```
```

Ou encore

```
```

17.4.1.2. Les colonnes

Vous savez déjà à l'avance que vous avez trois colonnes, dont la première contient un entier, la deuxième, une chaîne de caractères, et la troisième, un réel. Pour récupérer le contenu d'une de ces colonnes, il suffit d'utiliser une méthode du style `X getX(int columnIndex)` avec `X` le typage de la valeur à récupérer et `columnIndex` la colonne dans laquelle se trouve cette valeur. On peut par exemple récupérer un `Metier` complet avec :

```
```

Il ne vous est pas possible de récupérer le nom ou le type des colonnes, il vous faut donc le savoir à l'avance.

17.4.2. L'adaptateur pour les curseurs

Comme n'importe quel adaptateur, un `CursorAdapter` fera la transition entre des données et un `AdapterView`. Cependant, comme on trouve rarement *une seule* information dans un curseur, on préférera utiliser un `SimpleCursorAdapter`, qui est un équivalent au `SimpleAdapter` que nous avons déjà étudié.

Pour construire ce type d'adaptateur, on utilisera le constructeur suivant :

```
```

… où :

- `layout` est l'identifiant de la mise en page des vues dans l'`AdapterView`.
- `c` est le curseur. On peut mettre `null` si on veut ajouter le curseur *a posteriori*.
- `from` indique une liste de noms des colonnes afin de lier les données au layout.
- `to` contient les `TextView` qui afficheront les colonnes contenues dans `from`.

Tout cela est un peu compliqué à comprendre, je le conçois. Alors nous allons faire un layout spécialement pour notre table `Metier`.

Avant tout, sachez que pour utiliser un `CursorAdapter` ou n'importe quelle classe qui dérive de `CursorAdapter`, votre curseur *doit* contenir une colonne qui s'appelle `_id`. Si ce n'est pas le cas, vous n'avez bien entendu pas à recréer tout votre schéma, il vous suffit d'adapter vos requêtes pour que la colonne qui permet l'identification s'appelle `_id`, ce qui donne avec la requête précédente : `SELECT id as _id, intitule, salaire from Metier;`.

Le layout peut par exemple ressembler au code suivant, que j'ai enregistré dans `cursor_row.xml`.

```
```

Ensuite, pour utiliser le constructeur, c'est très simple, il suffit de faire :

```
```

— Android intègre au sein même de son système une base de données SQLite qu'il partage avec toutes les applications du système (avec des droits très spécifique à chacune pour qu'elle n'aille pas voir chez le voisin).
— La solution la plus évidente pour utiliser une base de données est de mettre en place une classe qui maitrisera les accès entre l'application et la base de données.
— En fonction des besoins de votre application, il est utile de mettre en place une série d'opérations usuelles accessibles à partir d'un DAO. Ces méthodes sont l'ajout, la suppression, la mise à jour et la sélection de données.
— Les curseurs sont des objets qui contiennent les résultats d'une recherche dans une base de données. Ce sont en fait des objets qui fonctionnent comme les tableaux que nous avons vus précédemment sur les chapitres des adaptateurs, ils contiennent les colonnes et les lignes qui ont été renvoyées par la requête.