16

Gestion des événements du cycle de vie d'une activité

Bien que cela puisse ressembler à un disque rayé, n'oubliez pas que les terminaux Android sont, avant tout, des téléphones. Pour cette raison, certaines activités sont plus importantes que d'autres – pour un utilisateur, pouvoir recevoir un appel est sûrement plus important que faire un sudoku. En outre, un téléphone possède généralement moins de mémoire qu'un ordinateur de bureau ou qu'un ordinateur portable.

Par conséquent, votre activité risque de se faire tuer parce que d'autres activités ont lieu et que le système a besoin de la mémoire qu'elle occupe. Il faut considérer tout cela comme l'équivalent Android du "cercle de la vie" – votre activité meurt afin que d'autres puissent vivre, etc. Vous ne pouvez donc pas supposer que votre application s'exécutera jusqu'à son terme, que ce soit de votre point de vue ou de celui de l'utilisateur.

C'est l'un des exemples – peut-être le plus important – de l'impact du cycle de vie d'une activité sur le déroulement de vos propres applications. Dans ce chapitre, nous présenterons les différents états qui composent le cycle de vie d'une activité et la façon de les gérer correctement.

L'activité de Schroedinger

En général, une activité se trouve toujours dans l'un des quatre états suivants :

- **Active.** L'activité a été lancée par l'utilisateur, elle s'exécute au premier plan. C'est à cet état que l'on pense quand on évoque le fonctionnement d'une activité.
- **En pause.** L'activité a été lancée par l'utilisateur, elle s'exécute et elle est visible, mais une notification ou un autre événement occupe une partie de l'écran. Pendant ce temps, l'utilisateur voit l'activité mais peut ne pas être capable d'interagir avec elle. Lorsqu'un appel téléphonique est reçu, l'utilisateur a l'opportunité de prendre cet appel ou de l'ignorer, par exemple.
- **Stoppée.** L'activité a été lancée par l'utilisateur, elle s'exécute mais est cachée par d'autres activités qui ont été lancées ou vers lesquelles le système a basculé. Votre application ne pourra rien présenter d'intéressant à l'utilisateur directement : elle ne peut passer que par une `Notification`.
- **Morte.** L'activité n'a jamais été lancée (le téléphone vient d'être réinitialisé, par exemple) ou elle a été tuée, probablement à cause d'un manque de mémoire.

Vie et mort d'une activité

Android fera appel à votre activité en considérant les transitions entre les quatre états que nous venons de présenter. Certaines transitions peuvent provoquer plusieurs appels à votre activité, *via* les méthodes présentées dans cette section ; parfois, Android tuera votre application sans l'appeler. Tout ceci est assez flou et sujet à modifications : c'est la raison pour laquelle vous devez consulter attentivement la documentation officielle d'Android en plus de cette section pour décider des événements qui méritent attention et de ceux que vous pouvez ignorer.

Notez que vous devez appeler les versions de la superclasse lorsque vous implémentez les méthodes décrites ici ; sinon Android peut lever une exception.

`onCreate()` et `onDestroy()`

Tous les exemples que nous avons vus jusqu'à maintenant ont implémenté `onCreate()` dans leurs sous-classes d'`Activity`.

Cette méthode sera appelée dans les trois cas suivants :

- Lorsque l'activité est lancée pour la première fois (après le redémarrage du système, par exemple), `onCreate()` est appelée avec le paramètre `null`.
- Si l'activité s'est exécutée, puis qu'elle a été tuée, `onCreate()` sera appelée avec, pour paramètre, le `Bundle` obtenu par `onSaveInstanceState()` (voir plus loin).

- Si l'activité s'est exécutée et que vous l'ayez configurée pour qu'elle utilise des ressources différentes en fonction des états du terminal (mode portrait ou mode paysage, par exemple), elle sera recréée et `onCreate()` sera donc appelée.

C'est dans cette méthode que vous configurez l'interface utilisateur et tout ce qui ne doit être fait qu'une seule fois, quelle que soit l'utilisation de l'activité.

À l'autre extrémité du cycle de vie, `onDestroy()` peut être appelée lorsque l'activité prend fin, soit parce qu'elle a appelé `finish()` (qui "finit" l'activité), soit parce qu'Android a besoin de mémoire et l'a fermée prématurément. `onDestroy()` peut ne pas être appelée si ce besoin de mémoire est urgent (la réception d'un appel téléphonique, par exemple) et que l'activité se terminera quoi qu'il en soit. Par conséquent, `onDestroy()` est essentiellement destinée à libérer les ressources obtenues dans `onCreate()`.

`onStart()`, `onRestart()` et `onStop()`

Une activité peut être placée au premier plan, soit parce qu'elle vient d'être lancée, soit parce qu'elle y a été mise après avoir été cachée (par une autre activité ou la réception d'un appel, par exemple).

Dans ces deux cas, c'est la méthode `onStart()` qui est appelée. `onRestart()` n'est invoquée que lorsque l'activité a été stoppée et redémarre.

Inversement, `onStop()` est appelée lorsque l'activité va être stoppée.

`onPause()` et `onResume()`

La méthode `onResume()` est appelée immédiatement avant que l'activité ne passe au premier plan, soit parce qu'elle vient d'être lancée, soit parce qu'elle est repartie après avoir été stoppée, soit après la fermeture d'une boîte de dialogue (ouverte par la réception d'un appel, par exemple). C'est donc un bon endroit pour reconstruire l'interface en fonction de ce qui s'est passé depuis que l'utilisateur l'a vue pour la dernière fois. Si votre activité interroge un service pour savoir s'il y a de nouvelles informations (de nouvelles entrées dans un flux RSS, par exemple), `onResume()` est le bon moyen de rafraîchir la vue courante et, si nécessaire, de lancer un thread en arrière-plan (*via* un `Handler`, par exemple) pour modifier cette vue.

Inversement, tout ce qui détourne l'utilisateur de votre activité – essentiellement l'activation d'une autre activité – provoquera l'appel d'`onPause()`. Vous pouvez profiter de cette méthode pour annuler tout ce que vous aviez fait dans `onResume()` : arrêter les threads en arrière-plan, libérer les ressources en accès exclusif que vous auriez pu prendre (l'appareil photo, par exemple), etc.

Lorsque `onPause()` a été appelée, Android se réserve le droit de tuer à tout moment le processus de l'activité. Par conséquent, vous ne devriez pas supposer que vous pourrez recevoir d'autre événement de la part de celle-ci.

L'état de grâce

Pour l'essentiel, les méthodes que nous venons de mentionner interviennent au niveau général de l'application (`onCreate()` relie les dernières parties de l'interface, `onPause()` ferme les threads en arrière-plan, etc.).

Cependant, Android a pour but de fournir une apparence de simplicité de fonctionnement. Autrement dit, bien que les activités puissent aller et venir en fonction des besoins mémoire, les utilisateurs ne devraient pas savoir ce qu'il se trame. Un utilisateur qui utilisait la calculette devrait donc retrouver le ou les nombres sur lesquels il travaillait lorsqu'il la réutilise après une absence – sauf s'il avait lui-même fermé la calculette.

Pour que tout cela fonctionne, les activités doivent donc pouvoir sauvegarder, rapidement et efficacement, l'état de l'instance de l'application qu'elles exécutent. En outre, comme elles peuvent être tuées à tout moment, les activités peuvent devoir sauvegarder cet état plus fréquemment qu'on ne pourrait le supposer. Réciproquement, une activité qui redémarre doit récupérer son état antérieur afin d'apparaître dans la situation où elle se trouvait précédemment.

La sauvegarde de l'état d'une instance est gérée par la méthode `onSaveInstanceState()`, qui fournit un objet `Bundle` dans lequel l'activité peut placer les données qu'elle souhaite (le nombre affiché par la calculette, par exemple). L'implémentation de cette méthode doit être rapide – n'essayez pas d'en faire trop : placez simplement les données dans le `Bundle` et quittez la méthode.

Vous pouvez récupérer l'état de l'instance dans les méthodes `onCreate()` et `onRestoreInstanceState()` : c'est vous qui décidez du moment d'appliquer cet état à votre activité – l'une ou l'autre de ces méthodes de rappel convient.

Partie III

Stockage de données, services réseaux et API

17

Utilisation des préférences

Android offre plusieurs moyens de stocker les données dont a besoin une activité. Le plus simple consiste à utiliser le système des préférences.

Les activités et les applications peuvent sauvegarder leurs préférences sous la forme de paires clés/valeurs qui persisteront entre deux appels. Comme leur nom l'indique, le but principal de ces préférences est de permettre la mémorisation d'une configuration choisie par l'utilisateur – le dernier flux consulté par le lecteur RSS, l'ordre de tri par défaut des listes, etc. Vous pouvez, bien sûr, les stocker comme vous le souhaitez du moment que les clés sont des chaînes (`String`) et les valeurs, des types primitifs (`boolean`, `String`, etc.).

Les préférences peuvent être propres à une activité ou partagées par toutes les activités d'une application. Elles peuvent même être partagées par les applications, bien que cela ne soit pas encore possible avec les versions actuelles d'Android.

Obtenir ce que vous voulez

Pour accéder aux préférences, vous pouvez :

- appeler la méthode `getPreferences()` à partir de votre activité pour accéder à ses préférences spécifiques ;
- appeler la méthode `getSharedPreferences()` à partir de votre activité (ou d'un autre `Context` de l'application) pour accéder aux préférences de l'application ;

- appeler la méthode `getDefaultSharedPreferences()` d'un objet `PreferencesManager` pour accéder aux préférences partagées qui fonctionnent de concert avec le système des préférences globales d'Android.

Les deux premiers appels prennent un mode de sécurité en paramètre – pour le moment, utilisez la valeur 0. `getSharedPreferences()` attend également le nom d'un ensemble de préférences : en réalité, `getPreferences()` appelle `getSharedPreferences()` en lui passant le nom de classe de votre activité en paramètre. `getDefaultSharedPreferences()` prend en paramètre le `Context` des préférences (c'est-à-dire votre `Activity`).

Toutes ces méthodes renvoient une instance de `SharedPreferences` qui fournit un ensemble de méthodes d'accès aux préférences par leurs noms. Ces méthodes renvoient un résultat du type adéquat (`getBoolean()`, par exemple, renvoie une préférence booléenne). Elles attendent également une valeur par défaut, qui sera renvoyée s'il n'existe pas de préférences ayant la clé indiquée.

Définir vos préférences

À partir d'un objet `SharedPreferences`, vous pouvez appeler `edit()` pour obtenir un "éditeur" pour les préférences. Cet objet dispose d'un ensemble de méthodes modificatrices à l'image des méthodes d'accès de l'objet parent `SharedPreferences`. Il fournit également les méthodes suivantes :

- `remove()` pour supprimer une préférence par son nom ;
- `clear()` pour supprimer toutes les préférences ;
- `commit()` pour valider les modifications effectuées *via* l'éditeur.

La dernière est importante : si vous modifiez les préférences avec l'éditeur et que vous n'appeliez pas `commit()`, les modifications disparaîtront lorsque l'éditeur sera hors de portée.

Inversement, comme les préférences acceptent des modifications en direct, si l'une des parties de votre application (une activité, par exemple) modifie des préférences partagées, les autres parties (tel un service) auront immédiatement accès à la nouvelle valeur.

Un mot sur le framework

À partir de sa version 0.9, Android dispose d'un framework de gestion des préférences qui ne change rien à ce que nous venons de décrire. En fait, il existe surtout pour présenter un ensemble cohérent de préférences aux utilisateurs, afin que les applications n'aient pas à réinventer la roue.

L'élément central de ce framework est encore une structure de données XML. Vous pouvez décrire les préférences de votre application dans un fichier XML stocké dans le répertoire `res/xml/` du projet. À partir de ce fichier, Android peut présenter une interface graphique pour manipuler ces préférences, qui seront ensuite stockées dans l'objet `SharedPreferences` obtenu par `getDefaultSharedPreferences()`.

Voici, par exemple, le contenu d'un fichier XML des préférences, extrait du projet `Prefs/Simple` :

```
<PreferenceScreen
  xmlns:android="http://schemas.android.com/apk/res/android">
  <CheckBoxPreference
    android:key="@string/checkbox"
    android:title="Preference case a cocher"
    android:summary="Cochez ou decochez" />
  <RingtonePreference
    android:key="@string/ringtone"
    android:title="Preference sonnerie"
    android:showDefault="true"
    android:showSilent="true"
    android:summary="Choisissez une sonnerie" />
</PreferenceScreen>
```

La racine de ce document XML est un élément `PreferenceScreen` (nous expliquerons plus loin pourquoi il s'appelle ainsi). Comme le montre ce fichier, `PreferenceScreen` peut contenir des définitions de préférences – des sous-classes de `Preference`, comme `CheckBoxPreference` ou `RingtonePreference`. Comme l'on pourrait s'y attendre, elles permettent, respectivement, de cocher une case et de choisir une sonnerie. Dans le cas de `RingtonePreference`, vous pouvez autoriser les utilisateurs à choisir la sonnerie par défaut du système ou "silence".

Laisser les utilisateurs choisir

Lorsque vous avez mis en place le fichier XML des préférences, vous pouvez utiliser une activité "quasi intégrée" pour permettre aux utilisateurs de faire leur choix. Cette activité est "quasi intégrée" car il suffit d'en créer une sous-classe, de la faire pointer vers ce fichier et de la lier au reste de votre application.

Voici, par exemple, l'activité `EditPreferences` du projet `Prefs/Simple` :

```
package com.commonsware.android.prefs;
import android.app.Activity;
import android.os.Bundle;
import android.preference.PreferenceActivity;
public class EditPreferences extends PreferenceActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    addPreferencesFromResource(R.xml.preferences);
  }
}
```

Comme vous pouvez le constater, il n'y a pas grand-chose à lire car il suffit d'appeler `addPreferencesFromResource()` en lui indiquant la ressource XML contenant les préférences.

Vous devrez également ajouter cette activité à votre fichier `AndroidManifest.xml` :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.commonsware.android.prefs">
    <application android:label="@string/app_name">
        <activity
            android:name=".SimplePrefsDemo"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name=".EditPreferences"
            android:label="@string/app_name">
        </activity>
    </application>
</manifest>
```

Voici le code de `SimplePrefsDemo`, qui permet d'appeler cette activité à partir d'un menu d'options :

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  menu.add(Menu.NONE, EDIT_ID, Menu.NONE, "Modifier Prefs")
      .setIcon(R.drawable.misc)
      .setAlphabeticShortcut('m');
  menu.add(Menu.NONE, CLOSE_ID, Menu.NONE, "Fermeture")
      .setIcon(R.drawable.eject)
      .setAlphabeticShortcut('f');
  return(super.onCreateOptionsMenu(menu));
}
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
    case EDIT_ID:
      startActivity(new Intent(this, EditPreferences.class));
      return(true);
    case CLOSE_ID:
      finish();
      return(true);
  }
  return(super.onOptionsItemSelected(item));
}
```

La Figure 17.1 montre l'interface de configuration des préférences de cette application.

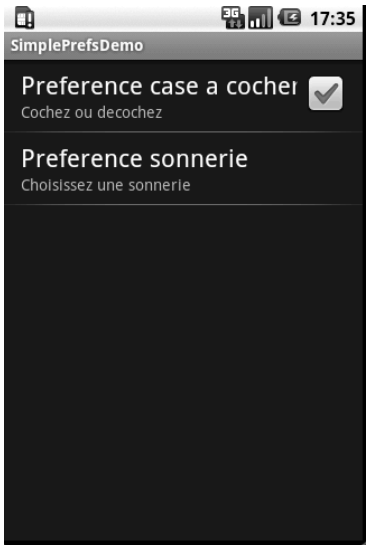

Figure 17.1
L'interface des préférences de `SimplePrefsDemo`.

La case à cocher peut être cochée ou décochée directement. Pour modifier la sonnerie, il suffit de cliquer sur son entrée dans les préférences pour voir apparaître une boîte de dialogue comme celle de la Figure 17.2.

Figure 17.2
Choix d'une sonnerie à partir des préférences.

Vous remarquerez qu'il n'existe pas de bouton "Save" ou "Commit" : les modifications sont sauvegardées dès qu'elles sont faites.

Outre ce menu, l'application `SimplePrefsDemo` affiche également les préférences courantes *via* un `TableLayout` :

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
>
  <TableRow>
    <TextView
        android:text="Case a cocher :"
        android:paddingRight="5px"
    />
    <TextView android:id="@+id/checkbox"
    />
  </TableRow>
  <TableRow>
    <TextView
        android:text="Sonnerie :"
        android:paddingRight="5px"
    />
    <TextView android:id="@+id/ringtone"
    />
  </TableRow>
</TableLayout>
```

Les champs de cette table se trouvent dans la méthode `onCreate()` :

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);
  checkbox=(TextView)findViewById(R.id.checkbox);
  ringtone=(TextView)findViewById(R.id.ringtone);
}
```

Ils sont mis à jour à chaque appel d'`onResume()` :

```
@Override
public void onResume() {
  super.onResume();
  SharedPreferences prefs=PreferenceManager
                          .getDefaultSharedPreferences(this);
  checkbox.setText(new Boolean(prefs
                          .getBoolean("checkbox", false))
                          .toString());
  ringtone.setText(prefs.getString("ringtone", "<unset>"));
}
```

Ceci signifie que les champs seront modifiés à l'ouverture de l'activité et après la fin de l'activité des préférences (*via* le bouton "back", par exemple).

La Figure 17.3 montre le contenu de cette table après le choix de l'utilisateur.

Figure 17.3
Liste des préférences de `SimplePrefsDemo`.

Ajouter un peu de structure

Si vous devez proposer un grand nombre de préférences à configurer, les mettre toutes dans une seule longue liste risque d'être peu pratique. Le framework d'Android permet donc de structurer un ensemble de préférences en *catégories* ou en *écrans*.

Les catégories sont créées à l'aide d'éléments `PreferenceCategory` dans le fichier XML des préférences et permettent de regrouper des préférences apparentées. Au lieu qu'elles soient toutes des filles de la racine `PreferenceScreen`, vous pouvez placer vos préférences dans les catégories appropriées en plaçant des éléments `PreferenceCategory` sous la racine. Visuellement, ces groupes seront séparés par une barre contenant le titre de la catégorie.

Si vous avez un très grand nombre de préférences – trop pour qu'il soit envisageable que l'utilisateur les fasse défiler –, vous pouvez également les placer dans des "écrans" distincts à l'aide de l'élément `PreferenceScreen` (le même que l'élément racine).

Tout fils de `PreferenceScreen` est placé dans son propre écran. Si vous imbriquez des `PreferenceScreen`, l'écran père affichera l'écran fils sous la forme d'une entrée : en la touchant, vous ferez apparaître le contenu de l'écran fils correspondant.

Le fichier XML des préférences du projet `Prefs/Structured` contient à la fois des éléments `PreferenceCategory` et des éléments `PreferenceScreen` imbriqués :

```
<PreferenceScreen
  xmlns:android="http://schemas.android.com/apk/res/android">
  <PreferenceCategory android:title="Preferences simples">
```

```
    <CheckBoxPreference
      android:key="@string/checkbox"
      android:title="Preference case a cocher"
      android:summary="Cochez ou decochez"
    />
    <RingtonePreference
      android:key="@string/ringtone"
      android:title="Preference sonnerie"
      android:showDefault="true"
      android:showSilent="true"
      android:summary="Choisissez une sonnerie"
    />
  </PreferenceCategory>
  <PreferenceCategory android:title="Ecrans">
    <PreferenceScreen
      android:key="detail"
      android:title="Ecran"
      android:summary="Preferences supplementaires dans une autre page">
      <CheckBoxPreference
        android:key="@string/checkbox2"
        android:title="Une autre case"
        android:summary="On ou Off. Peu importe."
      />
    </PreferenceScreen>
  </PreferenceCategory>
</PreferenceScreen>
```

Utilisé avec notre implémentation de `PreferenceActivity`, ce fichier XML produit une liste d'éléments comme ceux de la Figure 17.4, regroupés en catégories.

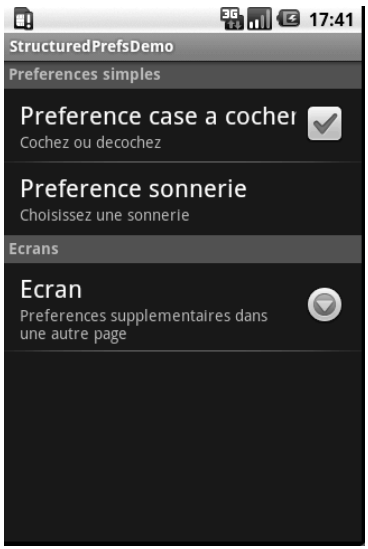

Figure 17.4
L'interface des préférences de `StructuredPrefsDemo`, montrant les catégories et un marqueur d'écran.

En touchant l'entrée du sous-écran, celui-ci s'affiche et montre les préférences qu'il contient (voir Figure 17.5).

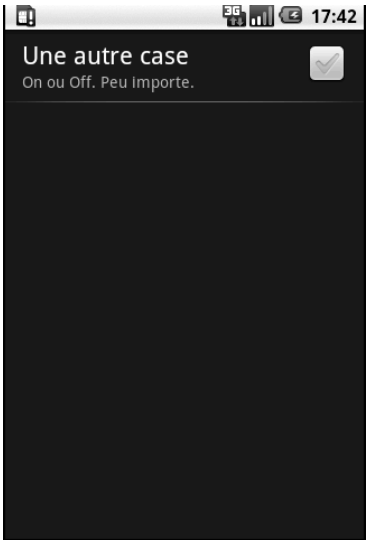

Figure 17.5
Le sous-écran de préférences de `StructuredPrefsDemo`.

Boîtes de dialogue

Toutes les préférences ne sont pas, bien sûr, que des cases à cocher et des sonneries.

Pour les autres, comme les champs de saisie et les listes, Android utilise des boîtes de dialogue. Les utilisateurs n'entrent plus directement leurs préférences dans l'activité interface des préférences mais touchent une préférence, remplissent une valeur et cliquent sur OK pour valider leur modification.

Comme le montre ce fichier XML, extrait du projet `Prefs/Dialogs`, les champs et les listes n'ont pas une structure bien différente de celle des autres types :

```
<PreferenceScreen
  xmlns:android="http://schemas.android.com/apk/res/android">
  <PreferenceCategory android:title="Simple Preferences">
    <CheckBoxPreference
      android:key="@string/checkbox"
      android:title="Preference case a cocher"
      android:summary="Cochez ou decochez"
    />
    <RingtonePreference
      android:key="@string/ringtone"
      android:title="Preference sonnerie"
      android:showDefault="true"
      android:showSilent="true"
```

```
      android:summary="Choisissez une sonnerie"
    />
  </PreferenceCategory>
  <PreferenceCategory android:title="Ecrans">
    <PreferenceScreen
      android:key="detail"
      android:title="Ecran"
      android:summary="Preferences supplementaires dans une autre page">
      <CheckBoxPreference
        android:key="@string/checkbox2"
        android:title="Autre case a cocher"
        android:summary="On ou Off. Peu importe."
      />
    </PreferenceScreen>
  </PreferenceCategory>
  <PreferenceCategory android:title="Preferences simples">
    <EditTextPreference
      android:key="@string/text"
      android:title="Dialogue de saisie d'un texte"
      android:summary="Cliquez pour ouvrir un champ de saisie"
      android:dialogTitle="Entrez un texte interessant"
    />
    <ListPreference
      android:key="@string/list"
      android:title="Dialogue de choix"
      android:summary="Cliquez pour ouvrir une liste de choix"
      android:entries="@array/villes"
      android:entryValues="@array/codes_aeroports"
      android:dialogTitle="Choisissez une ville" />
  </PreferenceCategory>
</PreferenceScreen>
```

Pour le champ de texte (`EditTextPreference`), outre le titre et le résumé de la préférence elle-même, nous donnons également un titre à la boîte de dialogue.

Pour la liste (`ListPreference`), on fournit à la fois un titre au dialogue et deux ressources de type tableau de chaînes : l'un pour les noms affichés, l'autre pour les valeurs correspondantes qui doivent être dans le même ordre – l'indice du nom affiché détermine la valeur stockée dans l'objet `SharedPreferences`. Voici par exemple les tableaux utilisés pour cette liste :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string-array name="villes">
    <item>Philadelphia</item>
    <item>Pittsburgh</item>
    <item>Allentown/Bethlehem</item>
    <item>Erie</item>
    <item>Reading</item>
```

```
    <item>Scranton</item>
    <item>Lancaster</item>
    <item>Altoona</item>
    <item>Harrisburg</item>
  </string-array>
  <string-array name="codes_aeroports">
    <item>PHL</item>
    <item>PIT</item>
    <item>ABE</item>
    <item>ERI</item>
    <item>RDG</item>
    <item>AVP</item>
    <item>LNS</item>
    <item>AOO</item>
    <item>MDT</item>
  </string-array>
</resources>
```

Comme le montre la Figure 17.6, les préférences comprennent désormais une catégorie supplémentaire contenant deux nouvelles entrées.

Figure 17.6
L'écran des préférences de `DialogsDemo`.

Toucher l'entrée "Dialogue de saisie d'un texte" provoque l'affichage d'un... dialogue de saisie d'un texte – ici, il est prérempli avec la valeur courante de la préférence (voir Figure 17.7).

Toucher l'entrée "Dialogue de choix" affiche... une liste de choix sous forme de boîte de dialogue présentant les noms des villes (voir Figure 17.8).

Figure 17.7
Modification d'une préférence avec un champ de saisie.

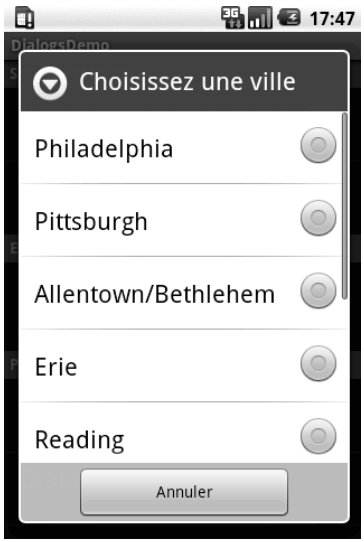

Figure 17.8
Modification d'une préférence avec une liste.

18

Accès aux fichiers

Bien qu'Android dispose de moyens de stockage structurés *via* les préférences et les bases de données, un simple fichier suffit parfois. Android offre donc deux modèles d'accès aux fichiers : l'un pour les fichiers fournis dans le paquetage de l'application, un autre pour ceux qui sont créés sur le terminal par l'application.

Allons-y !

Supposons que vous vouliez fournir des données statiques avec une application : une liste de mots pour un correcteur orthographique, par exemple. Le moyen le plus simple d'y parvenir consiste à placer ces données dans un fichier situé dans le répertoire `res/raw` : elles seront alors intégrées au fichier APK de l'application comme une ressource brute au cours du processus d'empaquetage.

Pour accéder à ce fichier, vous avez besoin d'un objet `Resources` que vous pouvez obtenir à partir de l'activité en appelant `getResources()`. Cet objet fournit la méthode `openRawResource()` pour récupérer un `InputStream` sur le fichier spécifié par son identifiant (un entier). Cela fonctionne exactement comme l'accès aux widgets avec `findViewById()` : si vous placez un fichier `mots.xml` dans `res/raw`, son identifiant dans le code Java sera `R.raw.mots`.

Comme vous ne pouvez obtenir qu'un `InputStream`, vous n'avez aucun moyen de modifier ce fichier : cette approche n'est donc vraiment utile que pour lire des données statiques. En outre, comme elles ne changeront pas jusqu'à ce que l'utilisateur installe une nouvelle version de votre paquetage, ces données doivent être valides pour une certaine période ou vous devrez fournir un moyen de les mettre à jour. Le moyen le plus simple de régler ce problème consiste à utiliser ces données pour construire une autre forme de stockage qui, elle, sera modifiable (une base de données, par exemple), mais cela impose d'avoir deux copies des données. Une autre solution consiste à les conserver telles quelles et à placer les modifications dans un autre fichier ou dans une base de données, puis à les fusionner lorsque vous avez besoin d'une vue complète de ces informations. Si votre application fournit une liste d'URL, par exemple, vous pourriez gérer un deuxième fichier pour stocker les URL ajoutées par l'utilisateur ou pour référencer celles qu'il a supprimées.

Le projet `Files/Static` reprend l'exemple `ListViewDemo` du Chapitre 8 en utilisant cette fois-ci un fichier XML à la place d'un tableau défini directement dans le programme. Le fichier de description XML est identique dans les deux cas :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent" >
  <TextView
    android:id="@+id/selection"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
  />
  <ListView
    android:id="@android:id/list"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:drawSelectorOnTop="false"
  />
</LinearLayout>
```

On a également besoin d'un autre fichier XML contenant les mots de la liste :

```
<words>
  <word value="lorem" />
  <word value="ipsum" />
  <word value="dolor" />
  <word value="sit" />
  <word value="amet" />
  <word value="consectetuer" />
  <word value="adipiscing" />
  <word value="elit" />
  <word value="morbi" />
  <word value="vel" />
```

```
  <word value="ligula" />
  <word value="vitae" />
  <word value="arcu" />
  <word value="aliquet" />
  <word value="mollis" />
  <word value="etiam" />
  <word value="vel" />
  <word value="erat" />
  <word value="placerat" />
  <word value="ante" />
  <word value="porttitor" />
  <word value="sodales" />
  <word value="pellentesque" />
  <word value="augue" />
  <word value="purus" />
</words>
```

Bien que cette structure XML ne soit pas exactement un modèle de concision, elle suffira pour notre démonstration.

Le code Java doit maintenant lire ce fichier, en extraire les mots et les placer quelque part pour que l'on puisse remplir la liste :

```
public class StaticFileDemo extends ListActivity {
  TextView selection;
  ArrayList<String> items=new ArrayList<String>();

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    selection=(TextView)findViewById(R.id.selection);

    try {
      InputStream in=getResources().openRawResource(R.raw.mots);
      DocumentBuilder builder=DocumentBuilderFactory
                              .newInstance()
                              .newDocumentBuilder();
      Document doc=builder.parse(in, null);
      NodeList words=doc.getElementsByTagName("word");

      for (int i=0;i<words.getLength();i++) {
        items.add(((Element)words.item(i)).getAttribute("value"));
      }

      in.close();
    }
```

```
      catch (Throwable t) {
        Toast
          .makeText(this, "Exception : " + t.toString(), 2000)
          .show();
      }

      setListAdapter(new ArrayAdapter<String>(this,
                                android.R.layout.simple_list_item_1,
                                items));
    }

    public void onListItemClick(ListView parent, View v, int position,
                    long id) {
      selection.setText(items.get(position).toString());
    }
  }
```

Les différences entre cet exemple et celui du Chapitre 8 se situent essentiellement dans le corps de la méthode `onCreate()`. Ici, on obtient un `InputStream` pour le fichier XML en appelant `getResources().openRawResource(R.raw.mots)`, puis on se sert des fonctionnalités d'analyse XML prédéfinies pour transformer le fichier en document DOM, en extraire les éléments `word` et placer les valeurs de leurs attributs `value` dans un objet `ArrayList` qui sera utilisé par l'`ArrayAdapter`.

Comme le montre la Figure 18.1, le résultat de l'activité est identique à celui l'exemple du Chapitre 8 car la liste de mots est la même.

Figure 18.1
L'application `StaticFileDemo`.

Comme nous le verrons au chapitre suivant, il existe évidemment des moyens encore plus simples d'utiliser les fichiers XML contenus dans le paquetage d'une application : utiliser une ressource XML, par exemple. Cependant, bien que cet exemple utilise XML, le fichier aurait pu simplement contenir un mot par ligne ou utiliser un format non reconnu nativement par le système de ressources d'Android.

Lire et écrire

Lire et écrire ses propres fichiers de données spécifiques est quasiment identique à ce que l'on pourrait faire avec une application Java traditionnelle. La solution consiste à utiliser les méthodes `openFileInput()` et `openFileOutput()` sur l'activité ou tout autre `Context` afin d'obtenir, respectivement, un `InputStream` et un `OutputStream`. À partir de là, le processus n'est pas beaucoup différent de celui des E/S Java classiques :

- On enveloppe ces flux selon les besoins, par exemple avec un `InputStreamReader` ou un `OutputStreamWriter` si l'on souhaite effectuer des E/S en mode texte.
- On lit ou on écrit les données.
- On libère le flux avec `close()` lorsqu'on a terminé.

Deux applications qui essaient de lire en même temps un fichier `notes.txt` *via* `openFileInput()` accéderont chacune à leur propre édition du fichier. Si vous voulez qu'un même fichier soit accessible à partir de plusieurs endroits, vous devrez sûrement créer un *fournisseur de contenu*, comme on l'explique au Chapitre 28. Notez également qu'`openFileInput()` et `openFileOutput()` ne prennent pas en paramètre des chemins d'accès (`chemin/vers/fichier.txt`, par exemple), mais uniquement des noms de fichiers.

Le code suivant, extrait du projet `Files/ReadWrite`, montre la disposition de l'éditeur de texte le plus simple du monde :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="vertical">
  <Button android:id="@+id/close"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Fermer" />
  <EditText
    android:id="@+id/editor"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:singleLine="false"
    />
</LinearLayout>
```

Les deux seuls éléments de l'interface sont un gros widget d'édition de texte et un bouton "Fermer" placé en dessous. Le code Java est à peine plus compliqué :

```
package com.commonsware.android.files;
import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;
import java.io.BufferedReader;
import java.io.File;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
public class ReadWriteFileDemo extends Activity {
  private final static String NOTES="notes.txt";
  private EditText editor;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    editor=(EditText)findViewById(R.id.editor);
    Button btn=(Button)findViewById(R.id.close);

    btn.setOnClickListener(new Button.OnClickListener() {
      public void onClick(View v) {
        finish();
      }
    });
  }

  public void onResume() {
    super.onResume();

    try {
      InputStream in=openFileInput(NOTES);

      if (in!=null) {
        InputStreamReader tmp=new InputStreamReader(in);
        BufferedReader reader=new BufferedReader(tmp);
        String str;
        StringBuffer buf=new StringBuffer();

        while ((str = reader.readLine()) != null) {
          buf.append(str + "\n");
        }
```

```
            in.close();
            editor.setText(buf.toString());
          }
        }
        catch (java.io.FileNotFoundException e) {
          // Ok, nous ne l'avons probablement pas encore créé
        }
        catch (Throwable t) {
          Toast
            .makeText(this, "Exception : "+ t.toString(), 2000)
            .show();
        }
      }

      public void onPause() {
        super.onPause();

        try {
          OutputStreamWriter out=
              new OutputStreamWriter(openFileOutput(NOTES, 0));

          out.write(editor.getText().toString());
          out.close();
        }
        catch (Throwable t) {
          Toast
            .makeText(this, "Exception : "+ t.toString(), 2000)
            .show();
        }
      }
    }
```

Nous commençons par lier le bouton à la fermeture de notre activité, en invoquant `finish()` sur l'activité à partir de `setOnClickListener()` lorsque nous cliquons sur le bouton.

Puis nous nous servons d'`onResume()` pour prendre le contrôle lorsque notre éditeur devient actif (après son lancement) ou le redevient (après avoir été figé). Nous lisons le fichier `notes.txt` à l'aide d'`openFileInput()` et nous y plaçons son contenu dans l'éditeur de texte. Si le fichier n'a pas été trouvé, nous supposons que c'est parce que c'est la première fois que l'activité s'exécute (ou que le fichier a été supprimé par d'autres moyens) et nous nous contentons de laisser l'éditeur vide.

Enfin, nous nous servons d'`onPause()` pour prendre le contrôle lorsque notre activité a été cachée par une autre ou qu'elle a été fermée (par notre bouton "Fermer", par exemple). Nous ouvrons alors le fichier `notes.txt` à l'aide d'`openFileOutput()` et nous y plaçons le contenu de l'éditeur de texte.

Le résultat est un bloc-notes persistant : tout ce qui est tapé le restera jusqu'à sa suppression ; le texte survivra à la fermeture de l'activité, à l'extinction du téléphone et aux autres situations similaires.