9

S'amuser avec les listes

L'humble `ListView` est l'un des widgets les plus importants et les plus utilisés d'Android. Que l'on choisisse un contact téléphonique, un courrier à faire suivre ou un ebook à lire, c'est de ce widget dont on se servira le plus souvent. Mais il serait évidemment plus agréable d'énumérer autre chose que du texte simple.

La bonne nouvelle est que les listes peuvent être aussi amusantes qu'on le souhaite... dans les limites de l'écran d'un mobile, évidemment. Cependant, cette décoration implique un peu de travail et met en œuvre certaines fonctionnalités d'Android que nous présenterons dans ce chapitre.

Premières étapes

Généralement, un widget `ListView` d'Android est une simple liste de texte – robuste mais austère. Cela est dû au fait que nous nous contentons de lui fournir un tableau de mots et que nous demandons à Android d'utiliser une disposition simple pour afficher ces mots sous forme de liste.

Cependant, vous pouvez également créer une liste d'icônes, d'icônes et de texte, de cases à cocher et de texte, etc. Tout cela dépend des données que vous fournissez à l'adaptateur et de l'aide que vous lui apportez pour créer un ensemble plus riche d'objets `View` pour chaque ligne.

Supposons, par exemple, que vous vouliez produire une liste dont chaque ligne est constituée d'une icône suivie d'un texte. Vous pourriez utiliser une disposition de ligne comme celle du projet `FancyLists/Static` :

```
<?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="wrap_content"
  android:orientation="horizontal">
  <ImageView
    android:id="@+id/icon"
    android:layout_width="22px"
    android:paddingLeft="2px"
    android:paddingRight="2px"
    android:paddingTop="2px"
    android:layout_height="wrap_content"
    android:src="@drawable/ok"
  />
  <TextView
    android:id="@+id/label"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="44sp"
  />
</LinearLayout>
```

On utilise ici un conteneur `LinearLayout` pour créer une ligne contenant une icône à gauche et un texte (utilisant une grande police agréable à lire) à droite.

Cependant, par défaut, Android ne sait pas que vous souhaitez utiliser cette disposition avec votre `ListView`. Pour établir cette connexion, vous devez donc indiquer à l'adaptateur l'identifiant de ressource de cette disposition personnalisée :

```
public class StaticDemo extends ListActivity {
  TextView selection;
  String[] items={"lorem", "ipsum", "dolor", "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
          "porttitor", "sodales", "pellentesque", "augue",
          "purus"};

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    setListAdapter(new ArrayAdapter<String>(this,
                        R.layout.row, R.id.label,
                        items));
```

```
    selection=(TextView)findViewById(R.id.selection);
  }

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

On peut remarquer que cette structure générale est identique à celle du projet `Selection/List` du Chapitre 8.

Le point essentiel de cet exemple est que l'on a indiqué à `ArrayAdapter` que l'on voulait utiliser notre propre disposition de ligne (`R.layout.row`) et que le `TextView` contenant le mot est désigné par `R.id.label` dans cette disposition. N'oubliez pas que, pour désigner une disposition (`row.xml`), il faut préfixer le nom de base du fichier de description par `R.layout` (`R.layout.row`).

On obtient ainsi une liste avec des icônes à droite. Ici, comme le montre la Figure 9.1, toutes les icônes sont les mêmes.

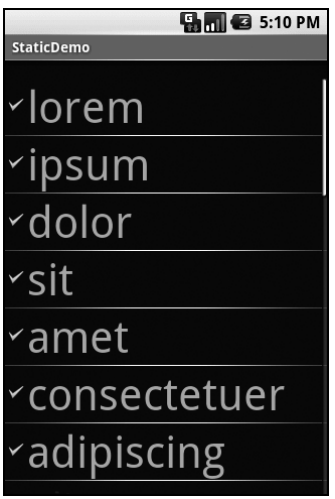

Figure 9.1
L'application `StaticDemo`.

Présentation dynamique

Cette technique – fournir une disposition personnalisée pour les lignes – permet de traiter très élégamment les cas simples, mais elle ne suffit plus pour les scénarios plus compliqués comme ceux qui suivent :

Chaque ligne utilise une disposition différente (certaines ont une seule ligne de texte, d'autres deux, par exemple).

Vous devez configurer chaque ligne différemment (par exemple pour mettre des icônes différentes en fonction des cas).

Dans ces situations, la meilleure solution consiste à créer une sous-classe de l'`Adapter` voulu, à redéfinir `getView()` et à construire soi-même les lignes. La méthode `getView()` doit renvoyer un objet `View` représentant la ligne située à la position fournie par l'adaptateur.

Reprenons par exemple le code précédent pour obtenir, grâce à `getView()`, des icônes différentes en fonction des lignes – une icône pour les mots courts, une autre pour les mots longs. Ce projet se trouve dans le répertoire `FancyLists/Dynamic` des exemples :

```
public class DynamicDemo extends ListActivity {
  TextView selection;
  String[] items={"lorem", "ipsum", "dolor", "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
          "porttitor", "sodales", "pellentesque", "augue",
          "purus"};

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

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

  class IconicAdapter extends ArrayAdapter {
    Activity context;

    IconicAdapter(Activity context) {
      super(context, R.layout.row, items);

      this.context=context;
    }

    public View getView(int position, View convertView,
                        ViewGroup parent) {
      LayoutInflater inflater=context.getLayoutInflater();
      View row=inflater.inflate(R.layout.row, null);
      TextView label=(TextView)row.findViewById(R.id.label);

      label.setText(items[position]);
```

```
        if (items[position].length()>4) {
          ImageView icon=(ImageView)row.findViewById(R.id.icon);

          icon.setImageResource(R.drawable.delete);
        }

        return(row);
      }
    }
  }
```

Le principe consiste à redéfinir `getView()` pour qu'elle renvoie une ligne dépendant de l'objet à afficher, qui est indiqué par l'indice `position` dans l'`Adapter`. Si vous examinez le code de cette implémentation, vous remarquerez que l'on utilise un objet `LayoutInflater`, ce qui mérite une petite explication.

Quelques mots sur l'inflation

Dans notre cas, "inflation" désigne le fait de convertir une description XML dans l'arborescence d'objets `View` qu'elle représente. Il s'agit indubitablement d'une partie de code assez ennuyeuse : on prend un élément, on crée une instance de la classe `View` appropriée ; on examine tous les attributs pour les convertir en propriétés, on parcourt tous les éléments fils et on recommence.

Heureusement, l'équipe qui a créé Android a encapsulé ce lourd traitement dans la classe `LayoutInflater`. Pour nos listes personnalisées, par exemple, nous voulons obtenir des `Views` pour chaque ligne de la liste et nous pouvons donc utiliser la notation XML pour décrire l'aspect des lignes.

Dans l'exemple précédent, nous transformons la description `R.layout.row` que nous avions créée dans la section précédente. Cela nous donne un objet `View` qui, en réalité, n'est autre que notre `LinearLayout` contenant un `ImageView` et un `TextView`, exactement comme cela est spécifié par `R.layout.row`. Cependant, au lieu de créer nous-mêmes tous ces objets et de les lier ensemble, le code XML et la classe `LayoutInflater` gèrent pour nous les "détails scabreux".

Revenons à nos moutons

Nous avons donc utilisé `LayoutInflater` pour obtenir un objet `View` représentant la ligne. Cette ligne est "vide" car le fichier de description statique ne sait pas quelles sont les données qu'elle recevra. Il vous appartient donc de la personnaliser et de la remplir comme vous le souhaitez avant de la renvoyer. C'est la raison pour laquelle :

On place le texte du label dans notre widget `label` en utilisant le mot situé à la position passée en paramètre à la méthode.

On regarde si ce mot fait plus de quatre caractères, auquel cas on recherche le widget `ImageView` de l'icône et on remplace sa ressource de base par une autre.

On dispose désormais d'une liste `ListView` contenant des icônes différentes, variant selon les entrées correspondantes de la liste (voir Figure 9.2).

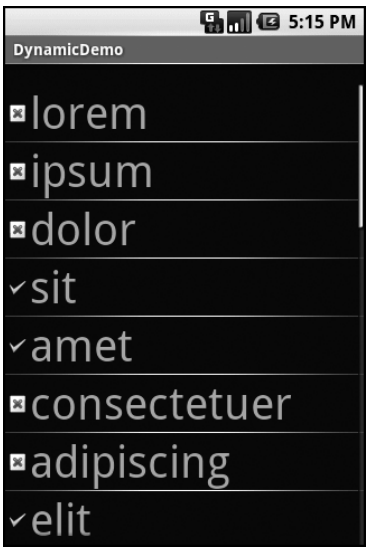

Figure 9.2
L'application `DynamicDemo`.

Il s'agit bien sûr d'un exemple assez artificiel, mais cette technique peut servir à personnaliser les lignes en fonction de n'importe quel critère – le contenu des colonnes d'un `Cursor`, par exemple.

Mieux, plus robuste et plus rapide

L'implémentation de `getView()` que nous venons de présenter fonctionne, mais elle est peu efficace. En effet, à chaque fois que l'utilisateur fait défiler l'écran, on doit créer tout un lot de nouveaux objets `View` pour les nouvelles lignes qui s'affichent. Le framework d'Android ne mettant pas automatiquement en cache les objets `View` existants, il faut en recréer de nouveaux, même pour des lignes que l'on avait créées très peu de temps auparavant. Ce n'est donc pas très efficace, ni du point de vue de l'utilisateur, qui risque de constater que la liste est lente, ni du point de vue de la batterie – chaque action du CPU consomme de l'énergie. Ce traitement supplémentaire est, par ailleurs, aggravé par la charge que l'on impose au ramasse-miettes (*garbage collector*) puisque celui-ci doit détruire tous les objets que l'on crée. Par conséquent, moins le code est efficace, plus la batterie du téléphone se décharge vite et moins l'utilisateur est content. On doit donc passer par quelques astuces pour éviter ces défauts.

Utilisation de `convertView`

La méthode `getView()` reçoit en paramètre un objet `View` nommé, par convention, `convertView`. Parfois, cet objet est `null`, auquel cas vous devez créer une nouvelle `View` pour la ligne (par inflation), comme nous l'avons expliqué plus haut.

Si `convertView` n'est pas `null`, en revanche, il s'agit en fait de l'une des `View` que vous avez déjà créées. Ce sera notamment le cas lorsque l'utilisateur fait défiler la `ListView` : à mesure que de nouvelles lignes apparaissent, Android tentera de réutiliser les vues des lignes qui ont disparu à l'autre extrémité, vous évitant ainsi de devoir les reconstruire totalement.

En supposant que chaque ligne ait la même structure de base, vous pouvez utiliser `findViewById()` pour accéder aux différents widgets qui composent la ligne, modifier leur contenu, puis renvoyer `convertView` à partir de `getView()` au lieu de créer une ligne totalement nouvelle.

Voici, par exemple, une écriture optimisée de l'implémentation précédente de `getView()`, extraite du projet `FancyLists/Recycling` :

```
public class RecyclingDemo extends ListActivity {
  TextView selection;
  String[] items={"lorem", "ipsum", "dolor", "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
          "porttitor", "sodales", "pellentesque", "augue",
          "purus"};

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

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

  class IconicAdapter extends ArrayAdapter {
    Activity context;

    IconicAdapter(Activity context) {
      super(context, R.layout.row, items);

      this.context=context;
    }
```

```
    public View getView(int position, View convertView,
                        ViewGroup parent) {
      View row=convertView;

      if (row==null) {
        LayoutInflater inflater=context.getLayoutInflater();

        row=inflater.inflate(R.layout.row, null);
      }

      TextView label=(TextView)row.findViewById(R.id.label);

      label.setText(items[position]);
      ImageView icon=(ImageView)row.findViewById(R.id.icon);

      if (items[position].length()>4) {
        icon.setImageResource(R.drawable.delete);
      }
      else {
        icon.setImageResource(R.drawable.ok);
      }

      return(row);
    }
  }
}
```

Si `convertView` est `null`, nous créons une ligne par inflation ; dans le cas contraire, nous nous contentons de la réutiliser. Le code pour remplir les contenus (image de l'icône, texte du label) est identique dans les deux cas. On évite ainsi une étape d'inflation potentiellement coûteuse lorsque `convertView` n'est pas `null`.

Cependant, cette approche ne fonctionne pas toujours. Si, par exemple, une `ListView` comprend des lignes ne contenant qu'une seule ligne de texte et d'autres en contenant plusieurs, la réutilisation des lignes existantes devient problématique car les layouts risquent d'être très différents. Si l'on doit créer une `View` pour une ligne qui compte deux lignes de texte, par exemple, on ne peut pas se contenter de réutiliser une `View` avec une seule ligne : il faut soit modifier les détails internes de cette `View`, soit l'ignorer et en créer une nouvelle.

Il existe bien entendu des moyens de gérer ce type de problème, comme rendre la seconde ligne de texte visible ou non en fonction des besoins, mais n'oubliez pas que chaque milliseconde d'utilisation du CPU d'un téléphone est précieuse – pour la fluidité de l'interface, mais surtout pour la batterie.

Ceci étant dit, surtout si vous débutez avec Android, intéressez-vous d'abord à obtenir la fonctionnalité que vous désirez et essayez ensuite d'optimiser les performances lors d'un

second examen de votre code. Ne tentez pas de tout régler d'un coup, sous peine de vous noyer dans un océan de `View`.

Utilisation du patron de conception "support"

L'appel de `findViewById()` est également coûteux : cette méthode plonge dans les lignes de la liste pour en extraire les widgets en fonction de leurs identifiants, afin que l'on puisse en personnaliser le contenu (pour modifier le texte d'un `TextView`, changer l'icône d'un `ImageView`, par exemple).

`findViewById()` pouvant trouver n'importe quel widget dans l'arbre des fils de la `View` racine de la ligne, cet appel peut demander un certain nombre d'instructions pour s'exécuter, notamment si l'on doit retrouver à nouveau des widgets que l'on a déjà trouvés auparavant.

Certains kits de développement graphiques évitent ce problème en déclarant les `View` composites, comme nos lignes, dans le code du programme (en Java, ici). L'accès aux différents widgets ne consiste plus, alors, qu'à appeler une méthode d'accès ou à lire un champ. Nous pourrions bien sûr faire de même avec Android, mais cela alourdirait le code. Nous préférons trouver un moyen de continuer à utiliser le fichier de description XML tout en mettant en cache les widgets fils essentiels de notre ligne, afin de ne devoir les rechercher qu'une seule fois.

C'est là qu'entre en jeu le patron de conception "support", qui est implémenté par une classe que nous appellerons `ViewWrapper`.

Tous les objets `View` disposent des méthodes `getTag()` et `setTag()`, qui permettent d'associer un objet quelconque au widget. Le patron "support" utilise ce "marqueur" pour détenir un objet qui, à son tour, détient chaque widget fils intéressant. En attachant le support à l'objet `View` de la ligne, on a accès immédiatement aux widgets fils qui nous intéressent à chaque fois que l'on utilise cette ligne, sans devoir appeler à nouveau `findViewById()`.

Examinons l'une de ces classes support (extrait du projet `FancyLists/ViewWrapper`) :

```
class ViewWrapper {
  View base;
  TextView label=null;
  ImageView icon=null;

  ViewWrapper(View base) {
    this.base=base;
  }

  TextView getLabel() {
    if (label==null) {
      label=(TextView)base.findViewById(R.id.label);
    }
```

```
    return(label);
  }

  ImageView getIcon() {
    if (icon==null) {
      icon=(ImageView)base.findViewById(R.id.icon);
    }

    return(icon);
  }
}
```

`ViewWrapper` ne détient pas seulement les widgets fils : elle les recherche uniquement si elle ne les détient pas déjà. Si vous créez un wrapper et que vous n'ayez jamais besoin d'un fils précis, il n'y aura donc jamais aucun appel de `findViewById()` pour le retrouver et vous n'aurez jamais à payer le prix de ces cycles CPU inutiles.

Le patron "support" permet également d'effectuer les traitements suivants :

Il regroupe au même endroit le transtypage de tous nos widgets, au lieu de le disséminer dans chaque appel à `findViewById()`.

Il permet de mémoriser d'autres informations sur les lignes, comme leur état, que nous ne voulons pas insérer dans le modèle sous-jacent.

L'utilisation de `ViewWrapper` consiste simplement à créer une instance de cette classe à chaque fois que l'on crée une ligne par inflation et à attacher cette instance à la vue de la ligne *via* `setTag()`, comme dans cette version de `getView()` :

```
public class ViewWrapperDemo extends ListActivity {
  TextView selection;
  String[] items={"lorem", "ipsum", "dolor", "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
          "porttitor", "sodales", "pellentesque", "augue",
          "purus"};

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

  private String getModel(int position) {
    return(((IconicAdapter)getListAdapter()).getItem(position));
  }
```

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

  class IconicAdapter extends ArrayAdapter<String> {
    Activity context;

    IconicAdapter(Activity context) {
      super(context, R.layout.row, items);

      this.context=context;
    }

    public View getView(int position, View convertView,
                        ViewGroup parent) {
      View row=convertView;
      ViewWrapper wrapper=null;

      if (row==null) {
        LayoutInflater inflater=context.getLayoutInflater();

        row=inflater.inflate(R.layout.row, null);
        wrapper=new ViewWrapper(row);
        row.setTag(wrapper);
      }
      else {
        wrapper=(ViewWrapper)row.getTag();
      }

      wrapper.getLabel().setText(getModel(position));

      if (getModel(position).length()>4) {
        wrapper.getIcon().setImageResource(R.drawable.delete);
      }
      else {
        wrapper.getIcon().setImageResource(R.drawable.ok);
      }

      return(row);
    }
  }
}
```

On teste si `convertView` est `null` pour créer au besoin les `View` de la ligne et l'on récupère (ou l'on crée) également le `ViewWrapper` de celle-ci. Accéder ensuite aux widgets fils consiste simplement à appeler les méthodes appropriées du wrapper.