6

Utilisation des widgets de base

Chaque kit de développement graphique possède des widgets de base : champs de saisie, labels, boutons, etc. Celui d'Android n'y fait pas exception et leur étude fournit une bonne introduction au fonctionnement des widgets dans les activités Android.

Labels

Le label (`TextView` pour Android) est le widget le plus simple. Comme dans la plupart des kits de développement, les labels sont des chaînes de textes non modifiables par les utilisateurs. Ils servent généralement à identifier les widgets qui leur sont adjacents ("Nom : ", par exemple, placé à côté d'un champ de saisie).

En Java, un label est une instance de la classe `TextView`. Cependant, ils seront le plus souvent créés dans les fichiers layout XML en ajoutant un élément `TextView` doté d'une propriété `android:text` pour définir le texte qui lui est associé. Si vous devez échanger des labels en fonction d'un certain critère – l'internationalisation, par exemple –, vous pouvez utiliser à la place une référence de ressource dans le code XML, comme nous

l'expliquerons au Chapitre 9. Un élément `TextView` possède de nombreux autres attributs, notamment :

- `android:typeface` pour définir le type de la police du label (monospace, par exemple) ;
- `android:textStyle` pour indiquer si le texte doit être en gras (`bold`), en italique (`italic`) ou les deux (`bold_italic`) ;
- `android:textColor` pour définir la couleur du texte du label, au format RGB hexadécimal (`#FF0000` pour un texte rouge, par exemple).

Voici le contenu du fichier de positionnement du projet `Basic/Label` :

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:text="Vous vous attendiez à quelque chose de plus profond ?"
  />
```

Comme le montre la Figure 6.1, ce fichier seul, avec le squelette Java fourni par la chaîne de production d'Android (`android create project`), produira l'application voulue.

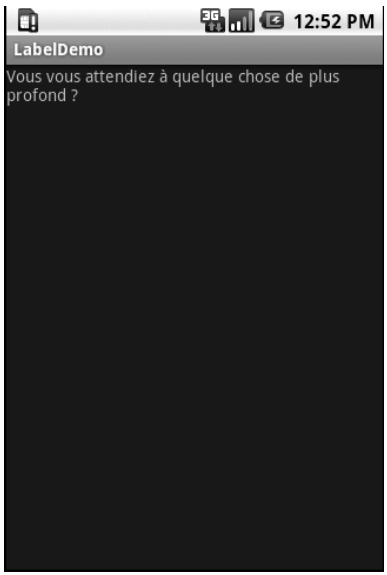


Figure 6.1
L'application `LabelDemo`.

Boutons

Nous avons déjà utilisé le widget `Button` au Chapitre 4. `Button` étant une sous-classe de `TextView`, tout ce qui a été dit dans la section précédente concernant le formatage du texte s'applique également au texte d'un bouton.

Images

Android dispose de deux widgets permettant d'intégrer des images dans les activités : `ImageView` et `ImageButton`. Comme leur nom l'indique, il s'agit, respectivement, des équivalents images de `TextView` et `Button`.

Chacun d'eux possède un attribut `android:src` permettant de préciser l'image utilisée. Cet attribut désigne généralement une ressource graphique (voir le chapitre consacré aux ressources). Vous pouvez également configurer le contenu de l'image en utilisant une URI d'un fournisseur de contenu, *via* un appel `setImageURI()`.

`ImageButton`, une sous-classe d'`ImageView`, lui ajoute les comportements d'un `Button` standard pour répondre aux clics et autres actions.

Examinons, par exemple, le contenu du fichier `main.xml` du projet `Basic/ImageView` :

```
<?xml version="1.0" encoding="utf-8"?>
<ImageView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/icon"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:adjustViewBounds="true"
    android:src="@drawable/molecule"
    />
```

Le résultat, qui utilise simplement l'activité produite automatiquement, est présenté à la Figure 6.2.

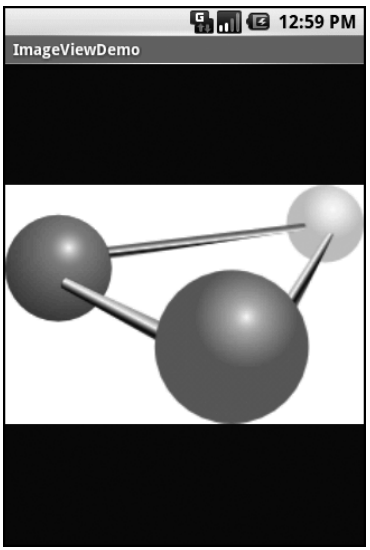


Figure 6.2
L'application `ImageViewDemo`.

Champs de saisie

Outre les boutons et les labels, les champs de saisie forment le troisième pilier de la plupart des outils de développement graphiques. Avec Android, ils sont représentés par le widget `EditText`, qui est une sous-classe de `TextView`, déjà vue pour les labels.

En plus des propriétés standard de `TextView` (`android:textStyle`, par exemple), `EditText` possède de nombreuses autres propriétés dédiées à la construction des champs. Parmi elles, citons :

- `android:autoText` pour indiquer si le champ doit fournir une correction automatique de l'orthographe.
- `android:capitalize` pour demander que le champ mette automatiquement en majuscule la première lettre de son contenu.
- `android:digits` pour indiquer que le champ n'acceptera que certains chiffres.
- `android:singleLine` pour indiquer si la saisie ne s'effectue que sur une seule ou plusieurs lignes (autrement dit, <Enter> vous place-t-il sur le widget suivant ou ajoute-t-il une nouvelle ligne ?).

Outre ces propriétés, vous pouvez configurer les champs pour qu'ils utilisent des méthodes de saisie spécialisées, avec les attributs `android:numeric` pour imposer une saisie uniquement numérique, `android:password` pour masquer la saisie d'un mot de passe et `android:phoneNumber` pour la saisie des numéros de téléphone. Pour créer une méthode de saisie particulière (afin, par exemple, de saisir des codes postaux ou des numéros de sécurité sociale), il faut implémenter l'interface `InputMethod` puis configurer le champ pour qu'il utilise cette méthode, à l'aide de l'attribut `android:inputMethod`.

Voici, par exemple, la description d'un `EditText` tiré du projet `Basic/Field` :

```
<?xml version="1.0" encoding="utf-8"?>
<EditText xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/field"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:singleLine="false"
  />
```

Vous remarquerez que la valeur d'`android:singleLine` est `false`, ce qui signifie que les utilisateurs pourront saisir plusieurs lignes de texte dans ce champ.

Le fichier `FieldDemo.java` de ce projet remplit le champ de saisie avec un peu de prose :

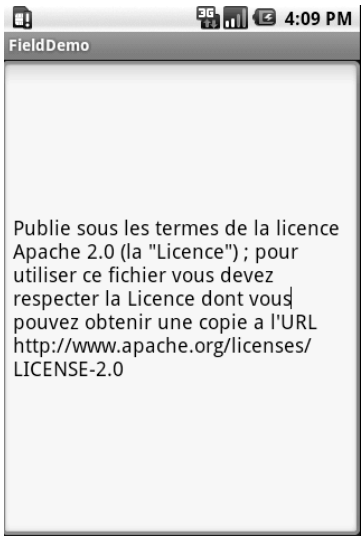
```
package com.commonsware.android.basic;
import android.app.Activity;
import android.os.Bundle;
```

```
import android.widget.EditText;
public class FieldDemo extends Activity {
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);

    EditText fld=(EditText)findViewById(R.id.field);
    fld.setText("Publie sous les termes de la licence Apache 2.0 " +
           "(la \"Licence\") ; pour utiliser ce fichier " +
            "vous devez respecter la Licence dont vous pouvez " +
            "obtenir une copie a l'URL " +
            "http://www.apache.org/licenses/LICENSE-2.0");
  }
}
```

La Figure 6.3 montre le résultat obtenu.

Figure 6.3
L'application `FieldDemo`.



L'émulateur d'Android n'autorise qu'une seule application d'un même paquetage Java dans le lanceur (application Launcher). Comme tous les exemples de ce chapitre appartiennent au paquetage `com.commonsware.android.basic`*, il n'apparaîtra qu'un seul exemple à la fois dans le lanceur.*

Certains champs offrent une complétion automatique afin de permettre à l'utilisateur d'entrer des informations sans taper l'intégralité du texte. Avec Android, ces champs sont des widgets `AutoCompleteTextView` ; ils seront présentés au Chapitre 8.

Cases à cocher

La case à cocher classique peut être dans deux états : cochée ou décochée. Un clic sur la case inverse son état pour indiquer un choix ("Livrer ma commande en urgence", par exemple). Le widget `CheckBox` d'Android permet d'obtenir ce comportement. Comme il dérive de la classe `TextView`, les propriétés de celles-ci comme `android:textColor` permettent également de formater ce widget.

Dans votre code Java, vous pouvez utiliser les méthodes suivantes :

- `isChecked()` pour savoir si la case est cochée ;
- `setChecked()` pour forcer la case dans l'état coché ou décoché ;
- `toggle()` pour inverser l'état de la case, comme si l'utilisateur avait cliqué dessus.

Vous pouvez également enregistrer un objet écouteur (il s'agira, ici, d'une instance d'`OnCheckedChangeListener`) pour être prévenu des changements d'état de la case.

Voici la déclaration XML d'une case à cocher, tirée du projet `Basic/CheckBox` :

```
<?xml version="1.0" encoding="utf-8"?>
<CheckBox xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/check"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Cette case est : decochee" />
```

Le fichier `CheckBoxDemo.java` correspondant récupère cette case à cocher et configure son comportement :

```
public class CheckBoxDemo extends Activity
  implements CompoundButton.OnCheckedChangeListener {
  CheckBox cb;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);

    cb=(CheckBox)findViewById(R.id.check);
    cb.setOnCheckedChangeListener(this);
  }

  public void onCheckedChanged(CompoundButton buttonView,
                               boolean isChecked) {
    if (isChecked) {
      cb.setText("Cette case est cochee");
    }
    else {
      cb.setText("Cette case est decochee");
    }
  }
}
```

Vous remarquerez que c'est l'activité qui sert d'écouteur pour les changements d'état de la case à cocher (avec `cb.setOnCheckedChangeListener(this)`) car elle implémente l'interface `OnCheckedChangeListener`.

La méthode de rappel de l'écouteur est `onCheckedChanged()` : elle reçoit la case qui a changé d'état et son nouvel état. Ici, on se contente de modifier le texte de la case pour refléter son état courant.

Cliquer sur la case modifie donc immédiatement son texte, comme le montrent les Figures 6.4 et 6.5.

Figure 6.4
L'application `CheckBoxDemo` avec la case décochée.

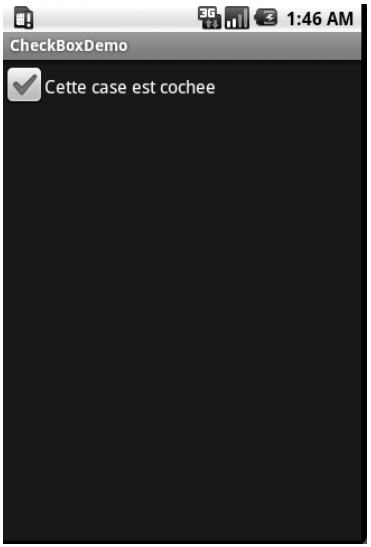


Figure 6.5
La même application avec la case cochée.

Boutons radio

Comme dans les autres outils de développement, les boutons radio d'Android ont deux états, telles les cases à cocher, mais peuvent être regroupés de sorte qu'un seul bouton radio puisse être coché par groupe à un instant donné.

Comme `CheckBox`, `RadioButton` hérite de la classe `CompoundButton`, qui dérive elle-même de `TextView`. Toutes les propriétés standard de `TextView` pour la police, le style, la couleur, etc. s'appliquent donc également aux boutons radio. Vous pouvez par conséquent appeler `isChecked()` sur un `RadioButton` pour savoir s'il est coché, `toggle()` pour le sélectionner, etc. exactement comme avec une `CheckBox`.

La plupart du temps, les widgets `RadioButton` sont placés dans un conteneur `RadioGroup` qui permet de lier les états des boutons qu'il regroupe afin qu'un seul puisse être sélectionné à un instant donné. En affectant un identifiant `android:id` au `RadioGroup` dans le fichier de description XML, ce groupe devient accessible au code Java, qui peut alors lui appliquer les méthodes suivantes :

- `check()` pour tester un bouton radio à partir de son identifiant (avec *`groupe`*`.check(R.id.radio1))` ;
- `clearCheck()` pour décocher tous les boutons du groupe ;
- `getCheckedRadioButtonId()` pour obtenir l'identifiant du bouton radio actuellement coché (cette méthode renvoie –1 si aucun bouton n'est coché).

Voici, par exemple, une description XML d'un groupe de boutons radio, tirée de l'exemple Basic/RadioButton :

```
<?xml version="1.0" encoding="utf-8"?>
<RadioGroup
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
    <RadioButton android:id="@+id/radio1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Caillou" />
    <RadioButton android:id="@+id/radio2"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Ciseaux" />
    <RadioButton android:id="@+id/radio3"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Papier" />
</RadioGroup>
```

La Figure 6.6 montre le résultat obtenu en utilisant le projet Java de base, fourni par Android.

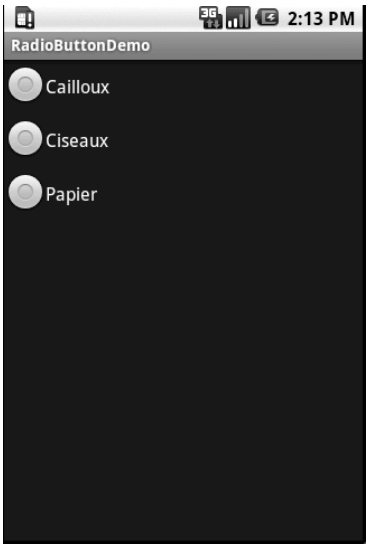


Figure 6.6
L'application `RadioButtonDemo`.

Vous remarquerez que, au départ, aucun bouton du groupe n'est coché. Pour que l'application sélectionne l'un de ces boutons dès son lancement, il faut appeler soit la méthode `setChecked()` sur le `RadioButton` concerné, soit la méthode `check()` sur le `RadioGroup` à partir de la méthode `onCreate()` de l'activité.

Résumé

Tous les widgets que nous venons de présenter dérivent de la classe `View` et héritent donc d'un ensemble de propriétés et de méthodes supplémentaires par rapport à celles que nous avons déjà décrites.

Propriétés utiles

Parmi les propriétés les plus utiles de `View`, citons :

- Le contrôle de la séquence de focus :
  - `android:nextFocusDown` ;
  - `android:nextFocusLeft` ;
  - `android:nextFocusRight` ;
  - `android:nextFocusUp`.

- `android:visibility`, qui contrôle la visibilité initiale du widget.
- `android:background`, qui permet de fournir au widget une couleur de fond au format RGB (`#00FF00` pour vert, par exemple).

Méthodes utiles

La méthode `setEnabled()` permet de basculer entre l'état actif et inactif du widget, alors que `isEnabled()` permet de tester si un widget est actif. On utilise souvent ces deux méthodes pour désactiver certains widgets en fonction de choix effectués à l'aide de `CheckBox` ou de `RadioButton`.

La méthode `requestFocus()` donne le focus à un widget et `isFocused()` permet de tester s'il a le focus. En utilisant les méthodes évoquées plus haut, on peut donc donner le focus à un widget précis après une opération de désactivation.

Les méthodes suivantes permettent de parcourir une arborescence de widgets et de conteneurs composant la vue générale d'une activité :

- `getParent()` renvoie le widget ou le conteneur parent.
- `findViewById()` permet de retrouver un widget fils d'après son identifiant.
- `getRootView()` renvoie la racine de l'arborescence (celle que vous avez fournie à l'activité *via* un appel à `setContentView()`).

7

Conteneurs

Les conteneurs permettent de disposer un ensemble de widgets (et, éventuellement, des conteneurs fils) pour obtenir la présentation de votre choix. Si, par exemple, vous préférez placer les labels à gauche et les champs de saisie à droite, vous aurez besoin d'un conteneur. Si vous voulez que les boutons OK et Annuler soient l'un à côté de l'autre, en bas à droite du formulaire, vous aurez également besoin d'un conteneur. D'un point de vue purement XML, si vous manipulez plusieurs widgets (le cas des `RadioButton` dans un `RadioGroup` est particulier), vous devrez utiliser un conteneur afin de disposer d'un élément racine dans lequel les placer.

La plupart des kits de développement graphiques utilisent des gestionnaires de disposition des widgets (*layout managers*) qui sont, le plus souvent, organisés sous forme de conteneurs. Java Swing, par exemple, dispose du gestionnaire `BoxLayout`, qui est utilisé avec certains conteneurs (comme `Box`). D'autres kits de développement, comme XUL et Flex, s'en tiennent strictement au modèle des boîtes, qui permet de créer n'importe quelle disposition *via* une combinaison adéquate de boîtes imbriquées.

Avec `LinearLayout`, Android offre également un modèle de disposition en boîtes, mais il fournit aussi un grand nombre de conteneurs autorisant d'autres systèmes de composition. Dans ce chapitre, nous étudierons trois conteneurs parmi les plus courants : `LinearLayout` (le modèle des boîtes), `RelativeLayout` (un modèle de positionnement relatif) et `TableLayout` (le modèle en grille) ; nous présenterons également `ScrollView`, un conteneur conçu pour faciliter la mise en place des conteneurs avec barres de défilement. Le chapitre suivant présentera d'autres conteneurs plus ésotériques.

Penser de façon linéaire

Comme on l'a déjà mentionné, `LinearLayout` est un modèle reposant sur des *boîtes* – les widgets ou les conteneurs fils sont alignés en colonnes ou en lignes, les uns après les autres, exactement comme avec `FlowLayout` en Java Swing, et `vbox` et `hbox` en Flex et XUL.

Avec Flex et XUL, la boîte est l'unité essentielle de disposition des widgets. Avec Android, vous pouvez utiliser `LinearLayout` exactement de la même façon, en vous passant des autres conteneurs. Obtenir la disposition que vous souhaitez revient alors principalement à identifier les imbrications et les propriétés des différentes boîtes – leur alignement par rapport aux autres boîtes, par exemple.

Concepts et propriétés

Pour configurer un `LinearLayout`, vous pouvez agir sur cinq paramètres : l'orientation, le modèle de remplissage, le poids, la gravité et le remplissage.

Orientation

L'orientation précise si le `LinearLayout` représente une ligne ou une colonne. Il suffit d'ajouter la propriété `android:orientation` à l'élément `LinearLayout` du fichier XML en fixant sa valeur à `horizontal` pour une ligne ou à `vertical` pour une colonne.

Cette orientation peut être modifiée en cours d'exécution en appelant la méthode `setOrientation()`et en lui fournissant en paramètre la constante `HORIZONTAL` ou `VERTICAL`.

Modèle de remplissage

Supposons que nous ayons une ligne de widgets – une paire de boutons radio, par exemple. Ces widgets ont une taille "naturelle" reposant sur celle de leur texte. Ces tailles combinées ne correspondent sûrement pas à la largeur de l'écran du terminal Android – notamment parce que les tailles des écrans varient en fonction des modèles. Il faut donc savoir que faire de l'espace restant.

Pour résoudre ce problème, tous les widgets d'un `LinearLayout` doivent fournir une valeur pour les propriétés `android:layout_width` et `android:layout_height`. Ces valeurs peuvent s'exprimer de trois façons différentes :

- Une dimension précise, comme 125 px, pour indiquer que le widget devra occuper exactement 125 pixels.
- `wrap_content`, pour demander que le widget occupe sa place naturelle sauf s'il est trop gros, auquel cas Android coupera le texte entre les mots pour qu'il puisse tenir.
- `fill_parent`, pour demander que le widget occupe tout l'espace disponible de son conteneur après que les autres widgets eurent été placés.

Les valeurs les plus utilisées sont les deux dernières, car elles sont indépendantes de la taille de l'écran ; Android peut donc ajuster la disposition pour qu'elle tienne dans l'espace disponible.

Poids

Que se passera-t-il si deux widgets doivent se partager l'espace disponible ? Supposons, par exemple, que nous ayons deux champs de saisie multilignes en colonne et que nous voulions qu'ils occupent tout l'espace disponible de la colonne après le placement de tous les autres widgets.

Pour ce faire, en plus d'initialiser `android:layout_width` (pour les lignes) ou `android:layout_height` (pour les colonnes) avec `fill_parent`, il faut également donner à `android:layout_weight`, une valeur qui indique la proportion d'espace libre qui sera affectée au widget. Si cette valeur est la même pour les deux widgets (1, par exemple), l'espace libre sera partagé équitablement entre eux. Si la valeur est 1 pour un widget et 2 pour l'autre, le second utilisera deux fois plus d'espace libre que le premier, etc.

Gravité

Par défaut, les widgets s'alignent à partir de la gauche et du haut. Si vous créez une ligne avec un `LinearLayout` horizontal, cette ligne commencera donc à se remplir à partir du bord gauche de l'écran.

Si ce n'est pas ce que vous souhaitez, vous devez indiquer une gravité à l'aide de la propriété `android:layout_gravity` d'un widget (ou en appelant la méthode `setGravity()` sur celui-ci) afin d'indiquer au widget et à son conteneur comment l'aligner par rapport à l'écran.

Pour une colonne de widgets, les gravités les plus courantes sont `left`, `center_horizontal` et `right` pour, respectivement, aligner les widgets à gauche, au centre ou à droite.

Pour une ligne, le comportement par défaut consiste à placer les widgets de sorte que leur texte soit aligné sur la ligne de base (la ligne invisible sur laquelle les lettres semblent reposer), mais il est possible de préciser une gravité `center_vertical` pour centrer verticalement les widgets dans la ligne.

Remplissage

Les widgets sont, par défaut, serrés les uns contre les autres. Vous pouvez augmenter l'espace intercalaire à l'aide de la propriété `android:padding` (ou en appelant la méthode `setPadding()` de l'objet Java correspondant au widget).

La valeur de remplissage précise l'espace situé entre le contour de la "cellule" du widget et son contenu réel. Elle est analogue aux marges d'un document dans un traitement de texte

– la taille de page peut être de 21 × 29,7 cm, mais des marges de 2 cm confinent le texte dans une surface de 19 × 27,7 cm.

La propriété `android:padding` permet de préciser le même remplissage pour les quatre côtés du widget ; son contenu étant alors centré dans la zone qui reste. Pour utiliser des valeurs différentes en fonction des côtés, utilisez les propriétés `android:paddingLeft`, `android:paddingRight`, `android:paddingTop` et `android:paddingBottom` (voir Figure 7.1).

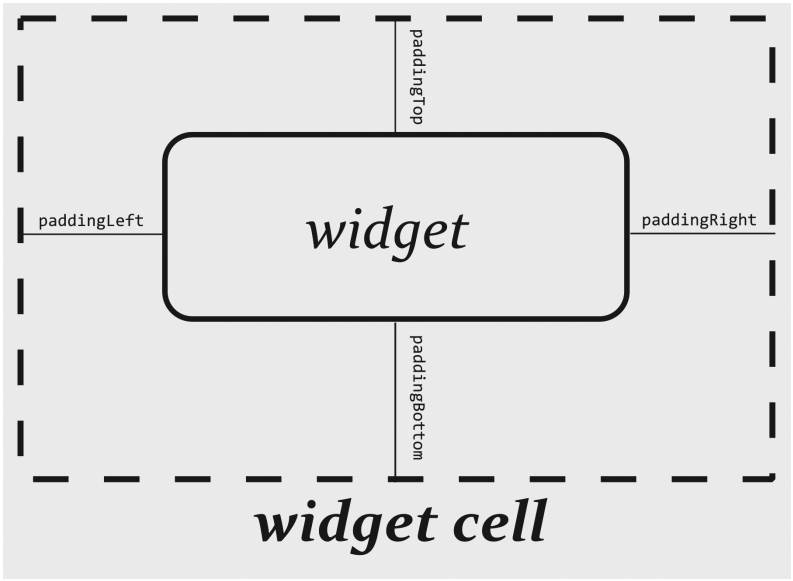


Figure 7.1
Relations entre un widget, sa cellule et ses valeurs de remplissage.

La valeur de ces propriétés est une dimension, comme `5px` pour demander un remplissage de 5 pixels.

Exemple

Voici le fichier de description XML de l'exemple `Containers/Linear`, qui montre les propriétés de l'élément `LinearLayout` :

```
<?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"
  >
  <RadioGroup android:id="@+id/orientation"
    android:orientation="horizontal"
    android:layout_width="wrap_content"
```

```
    android:layout_height="wrap_content"
    android:padding="5px">
    <RadioButton
      android:id="@+id/horizontal"
      android:text="horizontal" />
    <RadioButton
      android:id="@+id/vertical"
      android:text="vertical" />
  </RadioGroup>
  <RadioGroup android:id="@+id/gravity"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:padding="5px">
    <RadioButton
      android:id="@+id/left"
      android:text="gauche" />
    <RadioButton
      android:id="@+id/center"
      android:text="centre" />
    <RadioButton
      android:id="@+id/right"
      android:text="droite" />
  </RadioGroup>
</LinearLayout>
```

Vous remarquerez que le conteneur `LinearLayout` enveloppe deux `RadioGroup`. `RadioGroup` étant une sous-classe de `LinearLayout`, notre exemple revient donc à imbriquer des conteneurs `LinearLayout`.

Le premier élément `RadioGroup` configure une ligne (`android:orientation = "horizontal"`) de widgets `RadioButton`. Il utilise un remplissage de 5 pixels sur ses quatre côtés, afin de le séparer de l'autre `RadioGroup`. Sa largeur et sa hauteur valent toutes les deux `wrap_content` pour que les boutons radio n'occupent que l'espace dont ils ont besoin.

Le deuxième `RadioGroup` est une colonne (`android:orientation = "vertical"`) de trois `RadioButton`. Il utilise également un remplissage de 5 pixels sur tous ses côtés et sa hauteur est "naturelle" (`android:layout_height = "wrap_content"`). Cependant, sa propriété `android:layout_width` vaut `fill_parent`, ce qui signifie que la colonne de boutons radio "réclamera" toute la largeur de l'écran.

Pour ajuster ces valeurs en cours d'exécution en fonction de la saisie de l'utilisateur, il faut utiliser un peu de code Java :

```
package com.commonsware.android.containers;
import android.app.Activity;
import android.os.Bundle;
import android.view.Gravity;
```

```
import android.text.TextWatcher;
import android.widget.LinearLayout;
import android.widget.RadioGroup;
import android.widget.EditText;
public class LinearLayoutDemo extends Activity
  implements RadioGroup.OnCheckedChangeListener {
  RadioGroup orientation;
  RadioGroup gravity;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);

    orientation=(RadioGroup)findViewById(R.id.orientation);
    orientation.setOnCheckedChangeListener(this);
    gravity=(RadioGroup)findViewById(R.id.gravity);
    gravity.setOnCheckedChangeListener(this);
  }

  public void onCheckedChanged(RadioGroup group, int checkedId) {
    if (group==orientation) {
      if (checkedId==R.id.horizontal) {
        orientation.setOrientation(LinearLayout.HORIZONTAL);
      }
      else {
        orientation.setOrientation(LinearLayout.VERTICAL);
      }
    }
    else if (group==gravity) {
      if (checkedId==R.id.left) {
        gravity.setGravity(Gravity.LEFT);
      }
      else if (checkedId==R.id.center) {
        gravity.setGravity(Gravity.CENTER_HORIZONTAL);
      }
      else if (checkedId==R.id.right) {
        gravity.setGravity(Gravity.RIGHT);
      }
    }
  }
}
```

Dans `onCreate()`, nous recherchons nos deux conteneurs `RadioGroup` et nous enregistrons un écouteur pour chacun d'eux afin d'être prévenu du changement d'état des boutons radio (`setOnCheckedChangeListener(this)`). L'activité implémentant l'interface `OnCheckedChangeListener`, elle se comporte elle-même comme un écouteur.

Dans `onCheckedChanged()` (la méthode de rappel pour l'écouteur), on recherche le `RadioGroup` dont l'état a changé. S'il s'agit du groupe `orientation`, on ajuste l'orientation en fonction du choix de l'utilisateur. S'il s'agit du groupe `gravity`, on modifie la gravité.

La Figure 7.2 montre ce qu'affiche l'application lorsqu'elle est lancée dans l'émulateur.

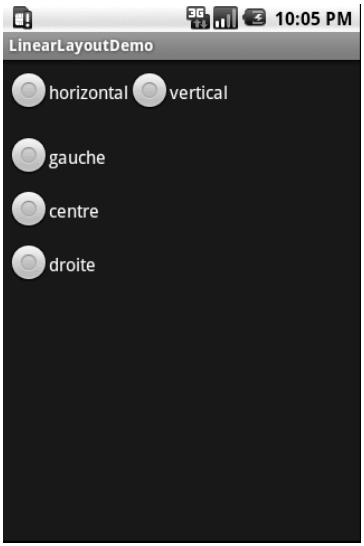


Figure 7.2
L'application `LinearLayoutDemo` lors de son lancement.

Si l'on clique sur le bouton `vertical`, le `RadioGroup` du haut s'ajuste en conséquence (voir Figure 7.3).

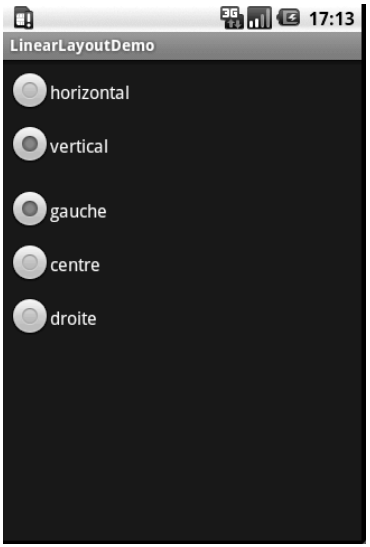


Figure 7.3
La même application, après avoir cliqué sur le bouton `vertical`.

Si l'on clique sur les boutons `centre` ou `droite`, le `RadioGroup` du bas s'ajuste également (voir Figures 7.4 et 7.5).

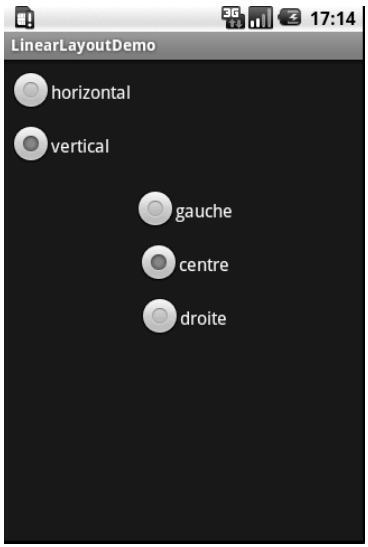


Figure 7.4
La même application, avec les boutons `vertical` et `centre` cochés.

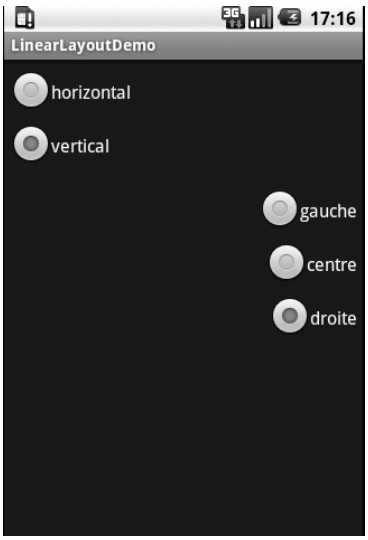


Figure 7.5
La même application, avec les boutons `vertical` et `droite` cochés.

Tout est relatif

Comme son nom l'indique, le `RelativeLayout` place les widgets relativement aux autres widgets du conteneur et de son conteneur parent. Vous pouvez ainsi placer le widget X en dessous et à gauche du widget Y ou faire en sorte que le bord inférieur du widget Z soit

aligné avec le bord inférieur du conteneur, etc. Ce gestionnaire de placement ressemble donc au conteneur `RelativeLayout`[1] de James Elliot pour Java Swing.

Concepts et propriétés

Il faut pouvoir faire référence à d'autres widgets dans le fichier de description XML et disposer d'un moyen d'indiquer leurs positions relatives.

Positions relatives à un conteneur

Les relations les plus simples à mettre en place sont celles qui lient la position d'un widget à celle de son conteneur :

- `android:layout_alignParentTop` précise que le haut du widget doit être aligné avec celui du conteneur.
- `android:layout_alignParentBottom` précise que le bas du widget doit être aligné avec celui du conteneur.
- `android:layout_alignParentLeft` précise que le bord gauche du widget doit être aligné avec le bord gauche du conteneur.
- `android:layout_alignParentRight` précise que le bord droit du widget doit être aligné avec le bord droit du conteneur.
- `android:layout_centerHorizontal` précise que le widget doit être centré horizontalement dans le conteneur.
- `android:layout_centerVertical` précise que le widget doit être centré verticalement dans le conteneur.
- `android:layout_centerInParent` précise que le widget doit être centré horizontalement et verticalement dans le conteneur.

Toutes ces propriétés prennent soit la valeur `true`, soit la valeur `false`.

Le remplissage du widget est pris en compte lors de ces alignements. Ceux-ci reposent sur la cellule globale du widget (c'est-à-dire sur la combinaison de sa taille naturelle et de son remplissage).

Notation relative dans les propriétés

Les propriétés restantes concernant `RelativeLayout` ont comme valeur l'identité d'un widget du conteneur. Pour ce faire :

1. Associez des identifiants (attributs `android:id`) à tous les éléments que vous aurez besoin de désigner, sous la forme `@+id/...`.

1. http://www.onjava.com/pub/a/onjava/2002/09/18/relativelayout.html.

2. Désignez un widget en utilisant son identifiant, privé du signe plus (`@id/...`).

Si, par exemple, le widget A est identifié par `@+id/widget_a`, le widget B peut le désigner dans l'une de ses propriétés par `@id/widget_a`.

Positions relatives aux autres widgets

Quatre propriétés permettent de contrôler la position d'un widget par rapport aux autres :

- `android:layout_above` indique que le widget doit être placé au-dessus de celui qui est désigné dans cette propriété.
- `android:layout_below` indique que le widget doit être placé sous celui qui est désigné dans cette propriété.
- `android:layout_toLeftOf` indique que le widget doit être placé à gauche de celui qui est désigné dans cette propriété.
- `android:layout_toRightOf` indique que le widget doit être placé à droite de celui qui est désigné dans cette propriété.

Cinq autres propriétés permettent de contrôler l'alignement d'un widget par rapport à un autre :

- `android:layout_alignTop` indique que le haut du widget doit être aligné avec le haut du widget désigné dans cette propriété.
- `android:layout_alignBottom` indique que le bas du widget doit être aligné avec le bas du widget désigné dans cette propriété.
- `android:layout_alignLeft` indique que le bord gauche du widget doit être aligné avec le bord gauche du widget désigné dans cette propriété.
- `android:layout_alignRight` indique que le bord droit du widget doit être aligné avec le bord droit du widget désigné dans cette propriété.
- `android:layout_alignBaseline` indique que les lignes de base des deux widgets doivent être alignées.

La dernière propriété de cette liste permet d'aligner des labels et des champs afin que le texte semble "naturel". En effet, les champs de saisie étant matérialisés par une boîte, contrairement aux labels, `android:layout_alignTop` alignerait le haut de la boîte du champ avec le haut du label, ce qui ferait apparaître le texte du label plus haut dans l'écran que le texte saisi dans le champ.

Si l'on souhaite que le widget B soit placé à droite du widget A, l'élément XML du widget B doit donc contenir `android:layout_toRight = "@id/widget_a"` (où `@id/widget_a` est l'identifiant du widget A).

Ordre d'évaluation

L'ordre d'évaluation complique encore les choses. En effet, Android ne lit qu'une seule fois le fichier XML et calcule donc en séquence la taille et la position de chaque widget. Ceci a deux conséquences :

- On ne peut pas faire référence à un widget qui n'a pas été défini plus haut dans le fichier.
- Il faut vérifier que l'utilisation de la valeur `fill_parent` pour `android:layout_width` ou `android:layout_height` ne "consomme" pas tout l'espace alors que l'on n'a pas encore défini tous les widgets.

Exemple

À titre d'exemple, étudions un "formulaire" classique, composé d'un champ, d'un label et de deux boutons, OK et Annuler.

Voici le fichier de disposition XML du projet `Containers/Relative` :

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:padding="5px">
<TextView android:id="@+id/label"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="URL:"
    android:paddingTop="15px"/>
  <EditText
    android:id="@+id/entry"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_toRightOf="@id/label"
    android:layout_alignBaseline="@id/label"/>
  <Button
    android:id="@+id/ok"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/entry"
    android:layout_alignRight="@id/entry"
    android:text="Ok" />
  <Button
    android:id="@+id/cancel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_toLeftOf="@id/ok"
    android:layout_alignTop="@id/ok"
    android:text="Annuler" />
</RelativeLayout>
```

Nous entrons d'abord dans l'élément `RelativeLayout`. Ici, on veut utiliser toute la largeur de l'écran (`android:layout_width = "fill_parent"`), n'utiliser que la hauteur nécessaire (`android:layout_height = "wrap_content"`), avec un remplissage de 5 pixels entre les limites du conteneur et son contenu (`android:padding = "5px"`).

Puis nous définissons un label assez basique, hormis son remplissage de 15 pixels (`android:padding = "15px"`), que nous expliquerons plus loin.

Nous ajoutons ensuite le champ que nous voulons placer à droite du label, avec sa ligne de base alignée avec celle du label et nous faisons en sorte qu'il occupe le reste de cette "ligne". Ces trois caractéristiques sont gérées par trois propriétés :

- `android:layout_toRight = "@id/label";`
- `android:layout_alignBaseline = "@id/label";`
- `android:layout_width = "fill_parent".`

Si nous n'avions pas utilisé le remplissage de 15 pixels pour le label, le haut du champ serait coupé à cause du remplissage de 5 pixels du conteneur lui-même. En effet, la propriété `android: layout_alignBaseline = "@id/label"` se contente d'aligner les lignes de base du label et du champ. Par défaut, le haut du label est aligné avec le haut de son parent, or il est plus petit que le champ puisque ce dernier est entouré d'une boîte. Le champ dépendant de la position du label qui a déjà été définie (puisqu'il apparaît avant lui dans le fichier XML), le champ serait trop haut et son bord supérieur serait rogné par le remplissage du conteneur.

Vous rencontrerez probablement ce genre de problème lorsque vous voudrez mettre en place des `RelativeLayout` pour qu'ils apparaissent exactement comme vous le souhaitez.

La solution à ce casse-tête consiste, comme nous l'avons vu, à ajouter 15 pixels de remplissage au-dessus du label, afin de le pousser suffisamment vers le bas pour que le champ ne soit pas coupé.

Voici quelques "solutions" qui ne marchent pas :

- Vous ne pouvez pas utiliser `android:layout_alignParentTop` sur le champ car il ne peut pas y avoir deux propriétés qui tentent en même temps de définir la position verticale du champ. Ici, `android:layout_alignParentTop` entrerait en conflit avec la propriété `android:layout_alignBaseline = "@id/label"`, qui apparaît plus bas, et c'est cette dernière qui l'emporterait. Vous devez donc soit aligner le haut, soit aligner les lignes de base, mais pas les deux.
- Vous ne pouvez pas d'abord définir le champ puis placer le label à sa gauche car on ne peut pas faire de "référence anticipée" vers un widget – il doit avoir été défini avant de pouvoir y faire référence.

Revenons à notre exemple. Le bouton OK est placé sous le champ (`android:layout_below = "@id/entry"`) et son bord droit est aligné avec le bord droit du champ

(`android:layout_alignRight = "@id/entry"`). Le bouton Annuler est placé à gauche du bouton OK (`android:layout_toLeft = "@id/ok"`) et son bord supérieur est aligné avec celui de son voisin (`android:layout_alignTop = "@id/ok"`).

La Figure 7.6 montre le résultat affiché dans l'émulateur lorsque l'on se contente du code Java produit automatiquement.

Figure 7.6
L'application `RelativeLayoutDemo`.

Tabula Rasa

Si vous aimez les tableaux HTML ou les feuilles de calcul, vous apprécierez le conteneur `TableLayout` d'Android car il vous permet de positionner les widgets dans une grille. Vous pouvez ainsi définir le nombre de lignes et de colonnes, les colonnes qui peuvent se réduire ou s'agrandir en fonction de leur contenu, etc.

`TableLayout` fonctionne de concert avec le conteneur `TableRow`. Alors que `TableLayout` contrôle le comportement global du conteneur, les widgets eux-mêmes sont placés dans un ou plusieurs `TableRow`, à raison d'un par ligne de la grille.

Concepts et propriétés

Pour utiliser ce conteneur, il faut savoir gérer les widgets en lignes et en colonnes, et traiter ceux qui sont placés à l'extérieur des lignes.

Placement des cellules dans les lignes

C'est vous, le développeur, qui déclarez les lignes en plaçant les widgets comme des fils d'un élément `TableRow`, lui-même fils d'un `TableLayout`. Vous contrôlez donc directement la façon dont apparaissent les lignes dans le tableau.

C'est Android qui détermine automatiquement le nombre de colonnes mais, en fait, vous le contrôlez de façon indirecte.

Il y aura au moins autant de colonnes qu'il y a de widgets dans la ligne la plus longue. S'il y a trois lignes, par exemple – une ligne avec deux widgets, une avec trois widgets et une autre avec quatre widgets –, il y aura donc au moins quatre colonnes.

Cependant, un widget peut occuper plusieurs colonnes si vous utilisez la propriété `android:layout_span` en lui précisant le nombre de colonnes sur lesquelles doit s'étendre le widget concerné. Cette propriété ressemble donc à l'attribut `colspan` utilisé dans les tableaux HTML :

```
<TableRow>
  <TextView android:text="URL:" />
  <EditText
    android:id="@+id/entry"
    android:layout_span="3"/>
</TableRow>
```

Avec ce fragment XML, le champ s'étendra sur trois colonnes.

Généralement, les widgets sont placés dans la première colonne disponible. Dans l'extrait précédent, par exemple, le label irait dans la première colonne (la colonne 0 car leur numérotation commence à 0) et le champ s'étendrait sur les trois colonnes suivantes (les colonnes 1 à 3). Vous pouvez également placer un widget sur une colonne précise en vous servant de la propriété `android:layout_column` et en lui indiquant le numéro de colonne voulu :

```
<TableRow>
  <Button
    android:id="@+id/cancel"
    android:layout_column="2"
    android:text="Annuler" />
  <Button android:id="@+id/ok" android:text="Ok" />
</TableRow>
```

Avec cet extrait, le bouton Annuler sera placé dans la troisième colonne (la colonne 2) et le bouton OK, dans la colonne disponible suivante, c'est-à-dire la quatrième.

Fils de `TableLayout` qui ne sont pas des lignes

Généralement, les seuls fils directs de `TableLayout` sont des éléments `TableRow`. Cependant, vous pouvez également placer des widgets entre les lignes. En ce cas, `TableLayout` se comporte un peu comme un conteneur `LinearLayout` ayant une orientation verticale. Les largeurs de ces widgets seront automatiquement fixées à `fill_parent` pour remplir le même espace que la ligne la plus longue.

Un cas d'utilisation de cette configuration consiste à se servir d'un widget `View` (`<View android:layout_height = "2px" android:background = "#0000FF" />`) pour créer une barre de séparation bleue de 2 pixels et de la même largeur que le tableau.

Réduire, étirer et refermer

Par défaut, la taille de chaque colonne sera la taille "naturelle" de son widget le plus large (en tenant compte des widgets qui s'étendent sur plusieurs colonnes). Parfois, cependant, cela ne donne pas le résultat escompté et il faut alors intervenir plus précisément sur le comportement de la colonne.

Pour ce faire, vous pouvez utiliser la propriété `android:stretchColumns` de l'élément `TableLayout`, dont la valeur peut être un seul numéro de colonne (débutant à zéro) ou une liste de numéros de colonnes séparés par des virgules. Ces colonnes seront alors étirées pour occuper tout l'espace disponible de la ligne, ce qui est utile lorsque votre contenu est plus étroit que l'espace restant.

Inversement, la propriété `android:shrinkColumns` de `TableLayout`, qui prend les mêmes valeurs, permet de réduire la largeur effective des colonnes en découpant leur contenu en plusieurs lignes (par défaut, le contenu des widgets n'est pas découpé). Cela permet d'éviter qu'un contenu trop long pousse certaines colonnes à droite de l'écran.

Vous pouvez également tirer parti de la propriété `android:collapseColumns` de `TableLayout`, en indiquant là aussi un numéro ou une liste de numéros de colonnes. Celles-ci seront alors initialement "refermées", ce qui signifie qu'elles n'apparaîtront pas, bien qu'elles fassent partie du tableau. À partir de votre programme, vous pouvez refermer ou réouvrir les colonnes à l'aide de la méthode `setColumnCollapsed()`du widget `TableLayout`. Ceci permet aux utilisateurs de contrôler les colonnes importantes qui doivent apparaître et celles qui peuvent être cachées car elles ne leur sont pas utiles.

En outre, les méthodes `setColumnStretchable()` et `setColumnShrinkable()` permettent respectivement de contrôler l'étirement et la réduction des colonnes en cours d'exécution.

Exemple

En combinant les fragments XML présentés dans cette section, on produit une grille de widgets ayant la même forme que celle de l'exemple `RelativeLayoutDemo`, mais avec

une ligne de séparation entre la ligne label/champ et celle des boutons (ce projet se trouve dans le répertoire `Containers/Table`) :

```
<?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"
  android:stretchColumns="1">
  <TableRow>
    <TextView
        android:text="URL:" />
    <EditText android:id="@+id/entry"
      android:layout_span="3"/>
  </TableRow>
  <View
    android:layout_height="2px"
    android:background="#0000FF" />
  <TableRow>
    <Button android:id="@+id/cancel"
      android:layout_column="2"
      android:text="Annuler" />
    <Button android:id="@+id/ok"
      android:text="Ok" />
  </TableRow>
</TableLayout>
```

On obtient alors le résultat montré à la Figure 7.7.

Figure 7.7
L'application `TableLayoutDemo`.

ScrollView

Les écrans des téléphones sont généralement assez petits, ce qui oblige les développeurs à employer quelques astuces pour présenter beaucoup d'informations dans un espace réduit. L'une de ces astuces consiste à utiliser le défilement, afin que seule une partie de l'information soit visible à un instant donné, le reste étant disponible en faisant défiler l'écran vers le haut ou vers le bas.

`ScrollView` est un conteneur qui fournit un défilement à son contenu. Vous pouvez donc utiliser un gestionnaire de disposition qui peut produire un résultat trop grand pour certains écrans et l'envelopper dans un `ScrollView` tout en continuant d'utiliser la logique de ce gestionnaire. L'utilisateur ne verra alors qu'une partie de votre présentation et aura accès au reste *via* des barres de défilement.

Voici par exemple un élément `ScrollView` qui enveloppe un `TableLayout` (ce fichier XML est extrait du répertoire `Containers/Scroll`) :

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content">
  <TableLayout
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:stretchColumns="0">
    <TableRow>
      <View
        android:layout_height="80px"
        android:background="#000000"/>
      <TextView android:text="#000000"
        android:paddingLeft="4px"
        android:layout_gravity="center_vertical" />
    </TableRow>
    <TableRow>
      <View
        android:layout_height="80px"
        android:background="#440000" />
      <TextView android:text="#440000"
        android:paddingLeft="4px"
        android:layout_gravity="center_vertical" />
    </TableRow>
    <TableRow>
      <View
        android:layout_height="80px"
        android:background="#884400" />
```

```
        <TextView android:text="#884400"
          android:paddingLeft="4px"
          android:layout_gravity="center_vertical" />
      </TableRow>
      <TableRow>
        <View
          android:layout_height="80px"
          android:background="#aa8844" />
        <TextView android:text="#aa8844"
          android:paddingLeft="4px"
          android:layout_gravity="center_vertical" />
      </TableRow>
      <TableRow>
        <View
          android:layout_height="80px"
          android:background="#ffaa88" />
        <TextView android:text="#ffaa88"
          android:paddingLeft="4px"
          android:layout_gravity="center_vertical" />
      </TableRow>
      <TableRow>
        <View
          android:layout_height="80px"
          android:background="#ffffaa" />
        <TextView android:text="#ffffaa"
          android:paddingLeft="4px"
          android:layout_gravity="center_vertical" />
      </TableRow>
      <TableRow>
        <View
          android:layout_height="80px"
          android:background="#ffffff" />
        <TextView android:text="#ffffff"
          android:paddingLeft="4px"
          android:layout_gravity="center_vertical" />
      </TableRow>
    </TableLayout>
  </ScrollView>
```

Sans le `ScrollView`, la grille occuperait au moins 560 pixels (sept lignes de 80 pixels chacune, selon la définition de l'élément `View`). Certains terminaux peuvent avoir des écrans capables d'afficher autant d'informations, mais la plupart seront plus petits. Le `ScrollView` permet alors de conserver la grille tout en en présentant qu'une partie à la fois.

La Figure 7.8 montre ce qu'affichera l'émulateur d'Android au lancement de l'activité.

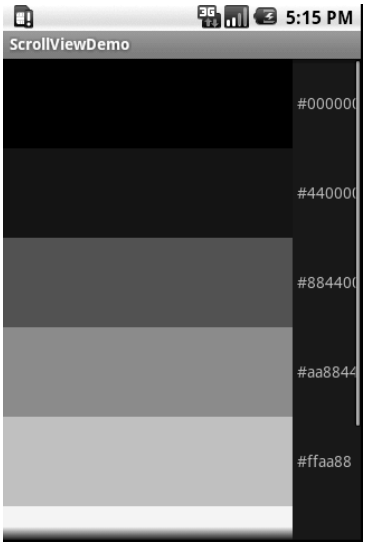


Figure 7.8
L'application `ScrollViewDemo`.

Vous remarquerez que l'on ne voit que cinq lignes, ainsi qu'une partie de la sixième. En pressant le bouton bas du pad directionnel, vous pouvez faire défiler l'écran afin de faire apparaître les lignes restantes. Vous remarquerez également que le bord droit du contenu est masqué par la barre de défilement – pour éviter ce problème, vous pourriez ajouter des pixels de remplissage sur ce côté.