14

Affichage de messages surgissant

Une activité (ou toute autre partie d'une application Android) a parfois besoin de s'exprimer.

Toutes les interactions avec les utilisateurs ne seront pas soignées, bien propres et contenues dans des activités composées de vues. Les erreurs surgissent ; les tâches en arrière-plan peuvent mettre plus de temps que prévu pour se terminer ; parfois, les événements, comme les messages entrants, sont asynchrones. Dans ce type de situation, vous pouvez avoir besoin de communiquer avec l'utilisateur en vous affranchissant des limites de l'interface classique.

Ce n'est évidemment pas nouveau : les messages d'erreur présentés dans des boîtes de dialogue existent depuis très longtemps. Cependant, il existe également des indicateurs plus subtils, allant des icônes apparaissant dans une barre des tâches (Windows) à des icônes animées dans un "dock" (Mac OS X), en passant par un téléphone qui vibre.

Android dispose de plusieurs moyens d'alerter les utilisateurs par d'autres systèmes que ceux des activités classiques. Les notifications, par exemple, sont intimement liées aux Intents et aux services et seront donc présentées au Chapitre 32. Nous étudierons ici deux méthodes permettant de faire surgir des messages : les toasts et les alertes.

Les toasts

Un toast est un message transitoire, ce qui signifie qu'il s'affiche et disparaît de lui-même, sans intervention de l'utilisateur. En outre, il ne modifie pas le focus de l'activité courante : si l'utilisateur est en train d'écrire le prochain traité fondamental sur l'art de la programmation, ses frappes au clavier ne seront donc pas capturées par le message.

Un toast étant transitoire, vous n'avez aucun moyen de savoir si l'utilisateur l'a remarqué. Vous ne recevrez aucun accusé de réception de sa part et le message ne restera pas affiché suffisamment longtemps pour ennuyer l'utilisateur. Un `Toast` est donc essentiellement conçu pour diffuser des messages d'avertissement – pour annoncer qu'une longue tâche en arrière-plan s'est terminée, que la batterie est presque vide, etc.

La création d'un toast est assez simple. La classe `Toast` fournit une méthode statique `makeText()` qui prend un objet `String` (ou un identifiant d'une ressource textuelle) en paramètre et qui renvoie une instance de `Toast`. Les autres paramètres de cette méthode sont l'activité (ou tout autre contexte) et une durée valant `LENGTH_SHORT` ou `LENGTH_LONG` pour exprimer la durée relative pendant laquelle le message restera visible.

Si vous préférez créer votre `Toast` à partir d'une `View` au lieu d'une simple chaîne ennuyeuse, il suffit de créer l'instance *via* le constructeur (qui attend un `Context`) puis d'appeler `setView()` pour lui indiquer la vue à utiliser et `setDuration()` pour fixer sa durée.

Lorsque le toast est configuré, il suffit d'appeler sa méthode `show()` pour qu'il s'affiche.

Les alertes

Si vous préférez utiliser le style plus classique des boîtes de dialogue, choisissez plutôt `AlertDialog`. Comme toute boîte de dialogue modale, un `AlertDialog` s'ouvre, prend le focus et reste affiché tant que l'utilisateur ne le ferme pas. Ce type d'affichage convient donc bien aux erreurs critiques, aux messages de validation qui ne peuvent pas être affichés correctement dans l'interface de base de l'activité ou à toute autre information dont vous voulez vous assurer la lecture immédiate par l'utilisateur.

Pour créer un `AlertDialog`, le moyen le plus simple consiste à utiliser la classe `Builder`, qui offre un ensemble de méthodes permettant de configurer un `AlertDialog`. Chacune de ces méthodes renvoie le `Builder` afin de faciliter le chaînage des appels. À la fin, il suffit d'appeler la méthode `show()` de l'objet `Builder` pour afficher la boîte de dialogue.

Voici les méthodes de configuration de `Builder` les plus utilisées :

- `setMessage()` permet de définir le "corps" de la boîte de dialogue à partir d'un simple message de texte. Son paramètre est un objet `String` ou un identifiant d'une ressource textuelle.

- `setTitle()` et `setIcon()` permettent de configurer le texte et/ou l'icône qui apparaîtra dans la barre de titre de la boîte de dialogue.
- `setPositiveButton()`, `setNeutralButton()` et `setNegativeButton()` permettent d'indiquer les boutons qui apparaîtront en bas de la boîte de dialogue, leur emplacement latéral (respectivement, à gauche, au centre ou à droite), leur texte et le code qui sera appelé lorsqu'on clique sur un bouton (en plus de refermer la boîte de dialogue).

Si vous devez faire d'autres configurations que celles proposées par `Builder`, appelez la méthode `create()` à la place de `show()` : vous obtiendrez ainsi une instance d'`AlertDialog` partiellement construite que vous pourrez configurer avant d'appeler l'une des méthodes `show()` de l'objet `AlertDialog` lui-même.

Après l'appel de `show()`, la boîte de dialogue s'affiche et attend une saisie de l'utilisateur.

Mise en œuvre

Voici le fichier de description XML du projet `Messages/Message` :

```
<?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" >
  <Button
    android:id="@+id/alert"
    android:text="Declencher une alerte"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"/>
  <Button
    android:id="@+id/toast"
    android:text="Lever un toast"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"/>
</LinearLayout>
```

Et voici son code Java :

```
public class MessageDemo extends Activity implements View.OnClickListener {
  Button alert;
  Button toast;
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
```

```
        setContentView(R.layout.main);

        alert=(Button)findViewById(R.id.alert);
        alert.setOnClickListener(this);
        toast=(Button)findViewById(R.id.toast);
        toast.setOnClickListener(this);
      }

      public void onClick(View view) {
        if (view==alert) {
          new AlertDialog.Builder(this)
            .setTitle("MessageDemo")
            .setMessage("Oups !")
            .setNeutralButton("Fermeture",
                              new DialogInterface.OnClickListener() {
              public void onClick(DialogInterface dlg, int sumthin) {
                // On ne fait rien - la boîte se fermera elle-même
              }
            })
            .show();
        }
        else {
          Toast
            .makeText(this, "<Clac, Clac>", Toast.LENGTH_SHORT)
            .show();
        }
      }
    }
```

Ce layout est tout ce qu'il y a de plus classique – il comprend simplement deux boutons pour déclencher l'alerte et le toast.

Lorsque nous cliquons sur le bouton "Déclencher une alerte", nous créons un `Builder` (avec `new Builder(this)`) pour configurer le titre (`setTitle("MessageDemo")`), le message (`setMessage("Oups !!")` et le bouton central (`setNeutralButton("Fermeture", new OnClickListener()` …) avant d'afficher la boîte de dialogue. Quand on clique sur ce bouton, la méthode de rappel `OnClickListener()` ne fait rien : le seul fait qu'on ait appuyé sur le bouton provoque la fermeture de la boîte de dialogue. Toutefois, vous pourriez mettre à jour des informations de votre activité en fonction de l'action de l'utilisateur, notamment s'il peut choisir entre plusieurs boutons. Le résultat est une boîte de dialogue classique, comme celle de la Figure 14.1.

Lorsque l'on clique sur le bouton "Lever un toast", la classe `Toast` crée un toast textuel (avec `makeText(this, "<Clac, Clac>", LENGTH_SHORT)`), puis l'affiche avec `show()`. Le résultat est un message de courte durée qui n'interrompt pas l'activité (voir Figure 14.2).

Figure 14.1
L'application `Message-Demo` *après avoir cliqué sur le bouton "Déclencher une alerte".*

Figure 14.2
La même application, après avoir cliqué sur le bouton "Lever un toast".

15

Utilisation des threads

Tout le monde souhaite que ses activités soient réactives. Répondre rapidement à un utilisateur (en moins de 200 millisecondes) est un bon objectif. Au minimum, il faut fournir une réponse en moins de 5 secondes ; sinon l'`ActivityManager` pourrait décider de jouer le rôle de la faucheuse et tuer votre activité car il considère qu'elle ne répond plus.

Mais, évidemment, votre programme peut devoir accomplir une tâche qui s'effectue en un temps non négligeable. Il y a deux moyens de traiter ce problème :

- réaliser les opérations coûteuses dans un service en arrière-plan en se servant de notifications pour demander aux utilisateurs de revenir à votre activité ;
- effectuer le travail coûteux dans un thread en arrière-plan.

Android dispose d'une véritable panoplie de moyens pour mettre en place des threads en arrière-plan tout en leur permettant d'interagir proprement avec l'interface graphique, qui, elle, s'exécute dans un thread qui lui est dédié. Parmi eux, citons les objets `Handler` et le postage d'objets `Runnable` à destination de la vue.

Les handlers

Le moyen le plus souple de créer un thread en arrière-plan avec Android consiste à créer une instance d'une sous-classe de `Handler`. Vous n'avez besoin que d'un seul objet `Handler` par activité et il n'est pas nécessaire de l'enregistrer manuellement ou quoi que ce soit d'autre – la création de l'instance suffit à l'enregistrer auprès du sous-système de gestion des threads.

Le thread en arrière-plan peut communiquer avec le `Handler`, qui effectuera tout son travail dans le thread de l'interface utilisateur de votre activité. C'est important car les changements de cette interface – la modification de ses widgets, par exemple – ne doivent intervenir que dans le thread de l'interface de l'activité.

Vous avez deux possibilités pour communiquer avec le `Handler` : les messages et les objets `Runnable`.

Les messages

Pour envoyer un `Message` à un `Handler`, appelez d'abord la méthode `obtainMessage()` afin d'extraire l'objet `Message` du pool. Cette méthode est surchargée pour vous permettre de créer un `Message` vide ou des messages contenant des identifiants de messages et des paramètres. Plus le traitement que doit effectuer le `Handler` est compliqué, plus il y a de chances que vous deviez placer des données dans le `Message` pour aider le `Handler` à distinguer les différents événements.

Puis envoyez le `Message` au `Handler` en passant par sa file d'attente des messages, à l'aide de l'une des méthodes de la famille `sendMessage...()` :

- `sendMessage()` place immédiatement le message dans la file.
- `sendMessageAtFrontOfQueue()` place immédiatement le message en tête de file (au lieu de le mettre à la fin, ce qui est le comportement par défaut). Votre message aura donc priorité par rapport à tous les autres.
- `sendMessageAtTime()` place le message dans la file à l'instant indiqué, qui s'exprime en millisecondes par rapport à l'*uptime* du système (`SystemClock.uptimeMillis()`).
- `sendMessageDelayed()` place le message dans la file après un certain délai, exprimé en millisecondes.

Pour traiter ces messages, votre `Handler` doit implémenter la méthode `handleMessage()`, qui sera appelée pour chaque message qui apparaît dans la file d'attente. C'est là que le handler peut modifier l'interface utilisateur s'il le souhaite. Cependant, il doit le faire rapidement car les autres opérations de l'interface sont suspendues tant qu'il n'a pas terminé.

Le projet `Threads/Handler`, par exemple, crée une `ProgressBar` et la modifie *via* un `Handler`. Voici son fichier de description XML :

```
<?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"
  >
  <ProgressBar android:id="@+id/progress"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" />
</LinearLayout>
```

La `ProgressBar`, qui a une largeur et une hauteur normales, utilise également la propriété `style`, qui ne sera pas décrite dans ce livre. Pour l'instant, il suffit de considérer qu'elle indique que la barre de progression devra être tracée horizontalement, en montrant la proportion de travail effectué.

Voici le code Java :

```
package com.commonsware.android.threads;
import android.app.Activity;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.widget.ProgressBar;
public class HandlerDemo extends Activity {
  ProgressBar bar;
  Handler handler=new Handler() {
    @Override
    public void handleMessage(Message msg) {
      bar.incrementProgressBy(5);
    }
  };
  boolean isRunning=false;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    bar=(ProgressBar)findViewById(R.id.progress);
  }

  public void onStart() {
    super.onStart();
    bar.setProgress(0);

    Thread background=new Thread(new Runnable() {
```

```
        public void run() {
          try {
            for (int i=0;i<20 && isRunning;i++) {
              Thread.sleep(1000);
              handler.sendMessage(handler.obtainMessage());
            }
          }
          catch (Throwable t) {
            // Termine le thread en arrière-plan
          }
        }
      });

      isRunning=true;
      background.start();
    }

    public void onStop() {
      super.onStop();
      isRunning=false;
    }
  }
```

Une partie de la construction de l'activité passe par la création d'une instance de `Handler` contenant notre implémentation de `handleMessage()`. Pour chaque message reçu, nous nous contentons de faire avancer la barre de progression de 5 points.

Dans la méthode `onStart()`, nous configurons un thread en arrière-plan. Dans une vraie application, celui-ci effectuerait une tâche significative mais, ici, nous nous mettons simplement en pause pendant 1 seconde, nous postons un `Message` au `Handler` et nous répétons ces deux opérations vingt fois. Combiné avec la progression de 5 points de la position de la `ProgressBar`, ce traitement fera donc avancer la barre à travers tout l'écran puisque la valeur maximale par défaut de la barre est 100. Vous pouvez ajuster ce maximum avec la méthode `setMax()` pour, par exemple, qu'il soit égal au nombre de lignes de la base de données que vous êtes en train de traiter et avancer de un la position pour chaque ligne.

Notez que l'on doit ensuite *quitter* `onStart()`. C'est un point crucial : la méthode `onStart()` est appelée dans le thread qui gère l'interface utilisateur de l'activité afin qu'elle puisse modifier les widgets et tout ce qui affecte cette interface – la barre de titre, par exemple. Ceci signifie donc que l'on doit sortir d'`onStart()`, à la fois pour laisser le `Handler` faire son travail et pour qu'Android ne pense pas que notre activité est bloquée.

La Figure 15.1 montre que l'activité se contente d'afficher une barre de progression horizontale.

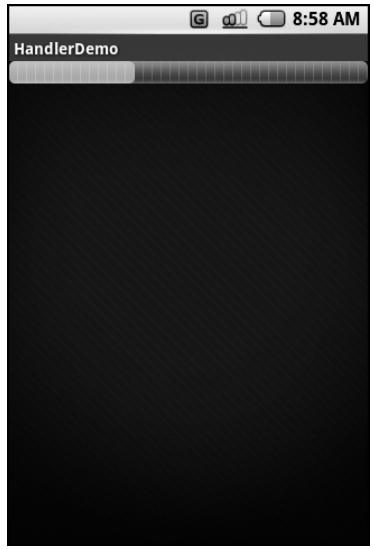

Figure 15.1
L'application `HandlerDemo`.

Les runnables

Si vous préférez ne pas vous ennuyer avec les objets `Message`, vous pouvez également passer des objets `Runnable` au `Handler`, qui les exécutera dans le thread de l'interface utilisateur. `Handler` fournit un ensemble de méthodes `post...()` pour faire passer les objets `Runnable` afin qu'ils soient traités.

Exécution sur place

Les méthodes `post()` et `postDelayed()` de `Handler`, qui permettent d'ajouter des objets `Runnable` dans la file d'attente des événements, peuvent également être utilisées avec les vues. Ceci permet de simplifier légèrement le code car vous pouvez alors vous passer d'un objet `Handler`. Toutefois, vous perdrez également un peu de souplesse. En outre, la classe `Handler` existe depuis longtemps dans Android et a probablement été mieux testée que cette technique.

Où est passé le thread de mon interface utilisateur ?

Parfois, vous pouvez ne pas savoir si vous êtes en train d'exécuter le thread de l'interface utilisateur de votre application. Si vous fournissez une partie de votre code sous la forme d'une archive JAR, par exemple, vous pouvez ne pas savoir si ce code est exécuté dans le thread de l'interface ou dans un thread en arrière-plan.

Pour résoudre ce problème, la classe `Activity` fournit la méthode `runOnUiThread()`. Elle fonctionne comme les méthodes `post()` de `Handler` et `View` car elle met dans une file un

`Runnable` pour qu'il s'exécute dans le thread de l'interface si vous n'y êtes pas déjà. Dans le cas contraire, elle appelle immédiatement le `Runnable`. Vous disposez ainsi du meilleur des deux mondes : aucun délai si vous êtes dans le thread de l'interface et aucun problème si vous n'y êtes pas.

Désynchronisation

Avec `AsyncTask`, Android 1.5 a introduit une nouvelle façon de concevoir les opérations en arrière-plan. Grâce à cette seule classe bien conçue, Android s'occupera du découpage du travail entre le thread de l'interface et un thread en arrière-plan. En outre, il allouera et désallouera lui-même le thread en arrière-plan et gérera une petite file de tâches, ce qui accentue encore le côté "prêt à l'emploi" d'`AsyncTask`.

La théorie

Il existe un dicton bien connu des vendeurs : "Lorsqu'un client achète un foret de 12 mm dans un magasin, il ne veut pas un foret de 13 mm : il veut percer des trous de 13 mm." Les magasins ne peuvent pas vendre les trous, ils vendent donc les outils (des perceuses et des forets) qui facilitent la création des trous.

De même, les développeurs Android qui se battent avec la gestion des threads en arrière-plan ne veulent pas vraiment des threads en arrière-plan : ils souhaitent qu'un certain travail s'effectue en dehors du thread de l'interface, afin que les utilisateurs ne soient pas bloqués et que les activités ne reçoivent pas la redoutable erreur "*application not responding*" (ANR). Bien qu'Android ne puisse pas, par magie, faire en sorte qu'un traitement ne consomme pas le temps alloué au thread de l'interface, il peut offrir des techniques permettant de faciliter et de rendre plus transparentes ces opérations en arrière-plan. `AsyncTask` en est un exemple.

Pour utiliser `AsyncTask`, il faut :

- créer une sous-classe d'`AsyncTask`, généralement sous la forme d'une classe interne à celle qui utilise la tâche (une activité, par exemple) ;
- redéfinir une ou plusieurs méthodes d'`AsyncTask` pour réaliser le travail en arrière-plan ainsi que toute opération associée à la tâche et qui doit s'effectuer dans le thread de l'interface (la mise à jour de la progression, par exemple) ;
- le moment venu, créer une instance de la sous-classe d'`AsyncTask` et appeler `execute()` pour qu'elle commence son travail.

Vous n'avez *pas* besoin :

- de créer votre propre thread en arrière-plan ;
- de terminer ce thread au moment voulu ;

- d'appeler des méthodes pour que des traitements s'effectuent dans le thread de l'interface.

AsyncTask, généricité et paramètres variables

La création d'une sous-classe d'`AsyncTask` n'est pas aussi simple que, par exemple, l'implémentation de l'interface `Runnable` car `AsyncTask` est une classe générique ; vous devez donc lui indiquer trois types de données :

- Le type de l'information qui est nécessaire pour le traitement de la tâche (les URL à télécharger, par exemple).
- Le type de l'information qui est passée à la tâche pour indiquer sa progression.
- Le type de l'information qui est passée au code après la tâche lorsque celle-ci s'est terminée.

En outre, les deux premiers types sont, en réalité, utilisés avec des paramètres variables, ce qui signifie que votre sous-classe d'`AsyncTask` les utilise *via* des tableaux.

Tout cela devrait devenir plus clair à l'étude d'un exemple.

Les étapes d'AsyncTask

Pour parvenir à vos fins, vous pouvez redéfinir quatre méthodes d'`AsyncTask`.

La seule que vous devez redéfinir pour que votre classe soit utilisable s'appelle `doInBackground()`. Elle sera appelée par `AsyncTask` dans un thread en arrière-plan et peut s'exécuter aussi longtemps qu'il le faut pour accomplir l'opération nécessaire à cette tâche spécifique. Cependant, n'oubliez pas que les tâches doivent avoir une fin – il est déconseillé d'utiliser `AsyncTask` pour réaliser une boucle infinie.

La méthode `doInBackground()` recevra en paramètre un tableau variable contenant des éléments du premier des trois types mentionnés ci-dessus – les type des données nécessaires au traitement de la tâche. Si la mission de cette tâche consiste, par exemple, à télécharger un ensemble d'URL, `doInBackground()` recevra donc un tableau contenant toutes ces URL.

Cette méthode doit renvoyer une valeur du troisième type de données mentionné – le résultat de l'opération en arrière-plan.

Vous pouvez également redéfinir `onPreExecute()`, qui est appelée à partir du thread de l'interface utilisateur avant que le thread en arrière-plan n'exécute `doInBackground()`. Dans cette méthode, vous pourriez par exemple initialiser une `ProgressBar` ou tout autre indicateur du début du traitement.

De même, vous pouvez redéfinir `onPostExecute()`, qui est appelée à partir du thread de l'interface graphique lorsque `doInBackground()` s'est terminée. Cette méthode reçoit en paramètre la valeur renvoyée par `doInBackground()` (un indicateur de succès ou d'échec,

par exemple). Vous pouvez donc utiliser cette méthode pour supprimer la `ProgressBar` et utiliser le travail effectué en arrière-plan pour mettre à jour le contenu d'une liste.

`onProgressUpdate()` est la quatrième méthode que vous pouvez redéfinir. Si `doInBackground()` appelle la méthode `publishProgress()` de la tâche, l'objet ou les objets passés à cette méthode seront transmis à `onProgressUpdate()`, mais dans le thread de l'interface. `onProgressUpdate()` peut donc alerter l'utilisateur que la tâche en arrière-plan a progressé – en mettant à jour une `ProgressBar` ou en continuant une animation, par exemple. Cette méthode reçoit un tableau variable d'éléments du second type mentionné plus haut, contenant les données publiées par `doInBackground()` *via* `publishProgress()`.

Exemple de tâche

Comme on l'a indiqué, l'implémentation d'une classe `AsyncTask` n'est pas aussi simple que celle d'une classe `Runnable`. Cependant, une fois passée la difficulté de la généricité et des paramètres variables, ce n'est pas si compliqué que cela.

Le projet `Threads/Asyncer` implémente une `ListActivity` qui utilise une `AsyncTask` :

```
package com.commonsware.android.async;
import android.app.ListActivity;
import android.os.AsyncTask;
import android.os.Bundle;
import android.os.SystemClock;
import android.widget.ArrayAdapter;
import android.widget.Toast;
import java.util.ArrayList;
public class AsyncDemo extends ListActivity {
  private static 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 savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
```

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

    new AddStringTask().execute();
  }

  class AddStringTask extends AsyncTask<Void, String, Void> {
    @Override
    protected Void doInBackground(Void... inutilise) {
      for (String item : items) {
        publishProgress(item);
        SystemClock.sleep(200);
      }

      return(null);
    }

    @Override
    protected void onProgressUpdate(String... item) {
      ((ArrayAdapter)getListAdapter()).add(item[0]);
    }

    @Override
    protected void onPostExecute(Void inutilise) {
      Toast
        .makeText(AsyncDemo.this, "Fini !", Toast.LENGTH_SHORT)
        .show();
    }
  }
}
```

Il s'agit d'une autre variante de la liste de mots *lorem ipsum* qui a déjà été souvent utilisée dans ce livre. Cette fois-ci, au lieu de simplement fournir la liste à un `ArrayAdapter`, on simule la création de ces mots dans un thread en arrière-plan à l'aide de la classe `AddStringTask`, notre implémentation d'`AsyncTask`.

Les sections qui suivent passent en revue les différentes parties de ce code.

Déclaration d'`AddStringTask`

```
class AddStringTask extends AsyncTask<Void, String, Void> {
```

On utilise ici la généricité pour configurer les types de données spécifiques dont nous aurons besoin dans `AddStringTask` :

- Nous n'avons besoin d'aucune information de configuration ; par conséquent, le premier type est `Void`.
- Nous voulons passer à `onProgressUpdate()` chaque chaîne "produite" par notre tâche en arrière-plan, afin de pouvoir l'ajouter à la liste. Le second type est donc `String`.
- Nous ne renvoyons pas de résultat à proprement parler (hormis les mises à jour) ; par conséquent, le troisième type est `Void`.

La méthode doInBackground()

```
@Override
protected Void doInBackground(Void... inutilise) {
  for (String item : items) {
    publishProgress(item);
    SystemClock.sleep(200);
  }
  return(null);
}
```

La méthode `doInBackground()` étant appelée dans un thread en arrière-plan, elle peut durer aussi longtemps qu'on le souhaite. Dans une vraie application, nous pourrions, par exemple, parcourir une liste d'URL et toutes les télécharger. Ici, nous parcourons notre liste statique de mots latins en appelant `publishProgress()` pour chacun d'eux, puis nous nous mettons en sommeil pendant le cinquième de seconde pour simuler un traitement.

Comme nous avons choisi de n'utiliser aucune information de configuration, nous n'avons normalement pas besoin de paramètre pour `doInBackground()`. Cependant, le contrat d'implémentation d'`AsyncTask` précise qu'il faut prendre un tableau variable d'éléments du premier type générique : c'est la raison pour laquelle le paramètre de cette méthode est `Void... inutilise`.

En outre, ayant choisi de ne rien renvoyer alors que le contrat stipule que nous devons renvoyer un objet du troisième type générique, le résultat de cette méthode est de type `Void` et nous renvoyons `null`.

La méthode onProgressUpdate()

```
@Override
protected void onProgressUpdate(String... item) {
  ((ArrayAdapter)getListAdapter()).add(item[0]);
}
```

La méthode `onProgressUpdate()` est appelée dans le thread de l'interface et nous voulons qu'elle signale à l'utilisateur que l'on est en train de charger les chaînes. Ici, nous ajoutons simplement la chaîne à l'`ArrayAdapter`, afin qu'elle soit ajoutée à la fin de la liste.

Cette méthode attend un tableau variable d'éléments du deuxième type générique (`String...` ici). Comme nous ne lui passons qu'une seule chaîne par appel à `publishProgress()`, on ne doit examiner que le premier élément du tableau `item`.

La méthode onPostExecute()

```
@Override
protected void onPostExecute(Void inutilise) {
  Toast
```

```
            .makeText(AsyncDemo.this, "Fini !", Toast.LENGTH_SHORT)
            .show();
    }
```

La méthode `onPostExecute()` est appelée dans le thread de l'interface et nous voulons qu'elle signale que l'opération qui s'exécutait en arrière-plan s'est terminée. Dans une vraie application, cela pourrait consister à supprimer une `ProgressBar` ou à stopper une animation, par exemple. Ici, nous nous contentons de "lever un toast".

Son paramètre est `Void inutilise` pour respecter le contrat d'implémentation.

L'activité

```
new AddStringTask().execute();
```

Pour utiliser `AddStringTask`, nous créons simplement une instance de cette classe et appelons sa méthode `execute()`. Ceci a pour effet de lancer la chaîne des événements qui font réaliser le travail par le thread en arrière-plan.

Si `AddStringTask` avait demandé des paramètres de configuration, nous n'aurions pas utilisé `Void` comme premier type générique et le constructeur aurait attendu zéro ou plusieurs paramètres de ce type. Ces valeurs auraient ensuite été passées à `doInBackground()`.

Le résultat

Comme le montre la Figure 15.2, cette activité affiche la liste qui se remplit "en temps réel" pendant quelques secondes, puis affiche un toast pour indiquer que le traitement est terminé.

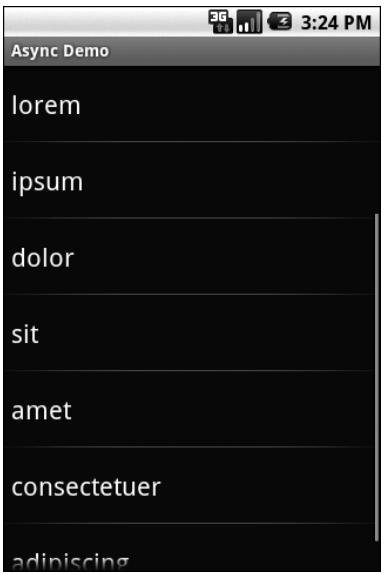

Figure 15.2
L'application `AsyncDemo`, au milieu du chargement de la liste des mots.

Éviter les pièges

Les threads en arrière-plan ne sont pas de mignons bébés bien sages, même en passant par le système des `Handler` d'Android. Non seulement ils ajoutent de la complexité, mais ils ont un coût réel en termes de mémoire, de CPU et de batterie.

C'est pour cette raison que vous devez tenir compte d'un grand nombre de scénarios lorsque vous les utilisez. Parmi eux :

- Les utilisateurs peuvent interagir avec l'interface graphique de votre activité pendant que le thread en arrière-plan s'essouffle. Si son travail est altéré ou modifié par une action de l'utilisateur, vous devez l'indiquer au thread. Les nombreuses classes du paquetage `java.util.concurrent` vous aideront dans cette tâche.
- L'activité peut avoir été supprimée alors que le travail en arrière-plan se poursuit. Après avoir lancé l'activité, l'utilisateur peut avoir reçu un appel téléphonique, suivi d'un SMS ; il a éventuellement ensuite effectué une recherche dans la liste de ses contacts. Toutes ces opérations peuvent suffire à supprimer votre activité de la mémoire. Le prochain chapitre présentera les différents événements par lesquels passe une activité ; vous devez gérer les bons et vous assurer de mettre correctement fin au thread en arrière-plan lorsque vous en avez l'occasion.
- Un utilisateur risque d'être assez mécontent si vous consommez beaucoup de temps CPU et d'autonomie de la batterie sans rien lui donner en retour. Tactiquement, ceci signifie que vous avez tout intérêt à utiliser une `ProgressBar` ou tout autre moyen de faire savoir à l'utilisateur qu'il se passe vraiment quelque chose. Stratégiquement, cela implique que vous devez être efficace – les threads en arrière-plan ne peuvent pas servir d'excuse à un code inutile et lent.
- Une erreur peut se produire au cours de l'exécution du traitement en arrière-plan – pendant que vous récupérez des informations à partir d'Internet, le terminal peut perdre sa connectivité, par exemple. La meilleure solution consiste alors à prévenir l'utilisateur du problème *via* une `Notification`, puis à mettre fin au thread en arrière-plan.